TURING 图灵程序设计丛书

MANNING



C# in Depth Third Edition

深入理解C#（第3版）

[英] Jon Skeet 著

姚琪琳 译

- 资深C# MVP扛鼎之作
- 深入理解语言特性，探究本源
- .NET开发人员必读经典

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



作者简介

Jon Skeet 谷歌软件工程师，微软资深C# MVP，拥有10余年C#项目开发经验。自2002年以来，他一直是C#社区、新闻组、国际会议和Stack Overflow网站上非常活跃的技术专家，回答了数以万计的C#和.NET相关问题。



译者简介

姚琪琳 具有多年.NET和Java开发经验，热爱C#，喜欢翻译和阅读。曾翻译过《精通C#（第6版）》《C#图解教程（第4版）》《C#与.NET 4高级程序设计（第5版）》等多本经典C#书籍。目前就职于ThoughtWorks。新浪微博：@珧麒麟。

TURING 图灵程序设计丛书



C# in Depth Third Edition

深入理解C#（第3版）

[英] Jon Skeet 著
姚琪琳 译

人民邮电出版社
北京

图书在版编目（C I P）数据

深入理解C# : 第3版 / (英) 斯基特 (Skeet,J.) 著;
姚琪琳 译. -- 北京 : 人民邮电出版社, 2014.4
(图灵程序设计丛书)
书名原文: C# in depth
ISBN 978-7-115-34642-1

Ⅰ. ①深… Ⅱ. ①斯… ②姚… Ⅲ. ①C语言—程序设
计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第027896号

内 容 提 要

本书是 C# 领域不可多得的经典著作。作者在详尽地展示 C# 各个知识点的同时，更注重从现象中挖掘本质。本书深入探索了 C# 的核心概念和经典特性，并将这些特性融入到代码中，让读者能够真正领会到 C# 之“深入”与“精妙”。在第 2 版的基础上，本书新增了 C# 5 的新特性——异步，并更新了随着技术的发展，已经不再适用的内容，确保整本书能达到读者期望的高标准。

如果你略微了解一点 C#，就可无障碍地阅读本书。

◆ 著　　　　[英] Jon Skeet
译　　　　姚琪琳
责任编辑　李　瑛
执行编辑　李　静　邢　妍
责任印制　焦志炜
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷
◆ 开本：800×1000　1/16
印张：32
字数：756千字　　　　2014年 4 月第 1 版
印数：1 – 4 000册　　　2014年 4 月北京第 1 次印刷
著作权合同登记号　图字：01-2013-7663号
定价：99.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

献给我的爱子Tom、 Robin和William。

对本书第1版的赞誉

总之，本书可以算是我读过的最好的计算机图书。

——Craig Pelkie，作家，System iNetwork

多年来我一直使用C#进行开发，但本书依然让我惊喜连连。它对委托、匿名方法和协变逆变的绝妙介绍让我印象特别深刻。即使你是一名经验丰富的开发者，本书仍然能让你学到C#语言中一些不为人知的东西。本书之“深入”，是其他书籍无法企及的。

——Adam J. Wolf，Southeast Valley .NET用户组

阅读本书是一大享受。它编排精妙，示例通俗易懂。我非常喜欢Lambda表达式这一章，并且很容易就被这一话题吸引。

——Jose Rolando Guay Paz，CSW Solutions公司Web开发者

作者将关于C#内部机理的丰富知识，汇集成了你手上这本文笔流畅、简洁实用的书。

——Jim Holmes，*Windows Developer Power Tools*作者

措辞严谨，示例精确，用最少的代码展示最全面的特性……阅读本书真是难得的享受啊！

——Franck Jeannin，Amazon评论员

如果你用C#进行了多年的开发，并且想了解一些内部原理，那么本书绝对适合你。

——Golo Roden，作家、演说家、.NET相关技术培训师

我所读过的最好的C#图书。

——Chris Mullins，C# MVP

对第2版的赞誉

一本关于C#的杰作。

——Kirill Osenkov，微软C#团队

如果你想精通C#，那么本书是必读之作。

——Tyson S. Maxwell，Raytheon资深软件工程师

我们打赌这是最好的C# 4图书。

——Nikander Bruggeman和Margriet Bruggeman，Lois & Clark IT Serivces的.NET顾问

对C# 4的独到见解实用且引人入胜。

——Joe Albahari，*LINQPad and C# 4.0 in a Nutshell*的作者

我所读过的最好的C#书籍之一。

——Aleksey Nudelman，C# Computing LLC的CEO

所有专业的C#开发者都应该阅读的书。

——Stuart Caborn，BNP Paribas资深开发者

C#所有主要版本中语言更新方面高度集中的、专家级的资源。对于所有想掌握C#语言最新动态的专业开发人员来说，本书必不可少。

——Sean Reilly，Point2 Technologies的程序员/分析师

为什么要一遍又一遍地阅读基础知识？Jon关注的是有嚼劲儿的新东西！

——Keith Hill，Agilent Technologies的软件架构师

所有你还没意识到需要掌握的C#知识。

——Jared Parsons，微软资深软件开发工程师

序

世上有两类钢琴家。

一类钢琴家弹琴并不是因为他们喜欢，而是因为父母强迫他们上钢琴课。另一类钢琴家弹琴是因为他们喜欢音乐，想创作音乐。他们不需要被强迫，相反，他们陶醉其中，时常忘记什么时候要停下来。

后一类人中，有人是把弹钢琴当作一种爱好。而有人则是为了生活，因此更需要投入、技巧和天赋。他们有一定的灵活性来选择弹奏哪些音乐流派和风格，不过这些选择主要还是由雇主的需要或者听众的口味来决定的。

后一类人中，有人主要就是为了钱，但也有一些专业人士即便没有报酬，也愿意在公共场合弹奏钢琴。他们喜欢运用自己的技巧和天赋为别人演奏音乐。在这个过程中，他们能找到许多乐趣。如果同时还有报酬，当然更是锦上添花。

后一类人中，有人是自学成材的，他们演奏乐曲是不看谱的。这些人有极高的天赋和能力，但除非通过音乐本身，否则无法向别人传递那种直观的感受。还有一些人无论在理论还是实践上都经过了正统的训练，他们能清楚地理解作曲家是用什么手法得到预期的情绪效果，并相应地改进自己的演绎手法。

后一类人中，有人从来没有打开钢琴看它的内部构造。还有一些人则对钢琴的发声原理好奇不已，最后发现是由于杠杆装置和绞盘在音锤敲击琴弦前的瞬间，牵引制音器的擒纵器，他们为弄明白由5 000 ~ 10 000个运动机件组成的这个乐器装置而感到高兴和自豪。

后一类人中，有人会对自己的手艺和成就心满意足，对它们带来的心灵上的愉悦和经济上的收入感到非常满意。但是，还有一些人不仅仅是艺术家、理论家和技师，他们会抽时间以导师的身份，将那些知识传授给其他人。

我不知道Jon Skeet是哪一类钢琴家。但是，我与这位微软C# MVP有多年的电子邮件交流，并经常看他的博客。我本人至少3遍逐字读完他的这本书，我清楚地知道Jon是后一种软件开发者：热情、博学、天资极高、有好奇心以及善于分析——是其他人的好老师。

C#是一种极为实用和快速发展的语言。通过添加查询能力、更丰富的类型推断、精简的匿名函数语法，等等，一种全新风格的编程语言已出现在我们的面前。与此同时，它代表的仍然是一种静态类型的、面向组件的开发方式，C#取得成功的立足之本没有变。

许多新元素会让人有矛盾的感觉。一方面，它们会显得比较“旧”（Lambda表达式可以追溯到20世纪上半叶计算机科学奠基的年代）。与此同时，对于那些习惯了现代面向对象编程的开发

者，它们又可能显得太新和太不熟悉。

Jon掌控了一切。对于需要理解C#最新版本“是什么”和“怎么做”的专业开发者，本书是理想的选择。此外，如果开发者还探索语言为什么要这样设计，从而加深他们对语言的理解，那么本书更是独一无二的。

为了利用语言提供的所有新能力，需要以全新的方式思考数据、函数以及它们之间的关系。这有点儿像经过多年的古典乐训练之后，开始尝试演奏爵士乐——或者相反。不管怎样，我期待下一代C#程序员能够“谱写”出优秀的乐章。祝你“谱曲”愉快，并感谢你选用了C#这个“主调”。

Eric Lippert
Coverity C#分析架构师

前　　言

哦，天哪。在撰写这篇前言时，我打开了第2版的前言，第一句话就是感觉距离撰写第1版的前言已经过去很长时间。对于现在来说，第2版是个遥远的记忆，而第1版则恍若隔世。我不知道是因为世界变化快，还是我不明白（记性差），但不管从哪方面来说，都值得静下来思考。

从第1版甚至是第2版到现在，发展格局已经发生了翻天覆地的变化。这里面有很多原因，而移动设备的崛起是最显而易见的因素。但很多挑战依旧没有改变。编写正确的国际化应用程序依然很难。在所有情况下优雅地处理错误依然很难。编写正确的多线程应用程序也依然很难，尽管多年来语言和库的改进已经大大地简化了这个任务。

最重要的是，在这篇前言的背景下，我认为开发人员对语言的掌握即使到了能够确定语言行为的程度，也仍需要了解他们正使用的这种语言。他们也许十分了解每个用到的API，甚至了解一些不经常使用的细枝末节[①]，但语言的核心应该是开发人员忠实的朋友，开发人员可以依靠它们按可预测的方式编写代码。

除了你所敲下的语言的字母，我相信理解它的精髓会有更大的好处。也许有时不管怎么努力，你都气得想砸键盘，但如果按照语言设计者的意图来组织代码，将获得前所未有的愉快体验。

① 我要说明：我对C#中的不安全代码和指针知之甚少。我完全不需要弄清楚它们。

关于封面插图

本书封面上的插图的标题是“音乐家”。插图来自一本土耳其奥斯曼帝国的服饰画册，由伦敦老邦德街的William Miller于1802年1月1日出版。画册的扉页已经丢失，因此我们很难推断准确的创作时间。此书的目录同时使用英语和法语标识插图，每张图片都有创作它的两位艺术家的名字，他们一定会为自己的作品出现在两百年后的一本计算机编程类图书的封面上而倍感惊讶。

Manning出版社的一个编辑在位于曼哈顿西26街“Garage”的古董跳蚤市场买到了这本画册。卖主是住在土耳其安卡拉的一个美国人，交易时间是在那天他准备收摊的时候。这位编辑没带够买这本画册所需的现金，并且卖主礼貌地拒绝了他使用信用卡和支票。卖主当天晚上要飞回安卡拉，看起来好像没什么希望了。该怎么解决呢？两个人最后通过握手约定的老式君子协议解决了。卖主提议通过银行转账付款，编辑在纸上抄下了收款银行的信息，随后画册就到他手里了。不用说，第二天我们就把款付给了卖主。我们感谢这位陌生人能如此信任我们的同事。这让我们回忆起了那个很久以前的美好时代。

我们Manning人崇尚创造性和主动性，而以两个世纪以前的丰富多彩的地区生活作为图书封面的素材，使得计算机商业充满趣味性，这本画册中的这张图片，把我们带到了那时的生活中。

致　　谢

你可能认为组织这本书的第3版（新增了两章）十分简单。的确，编写第15章和第16章的“绿色内容”[①]非常容易。但真正的工作远不止这些。我需要全书调整语言的细节，检查当年没问题现在却没道理的部分，还要确保整书能达到读者期望的高标准。幸好，有一群人支持我，让本书没有走改旗易帜的邪路。

最重要的是，我的家人一如既往地令人赞叹。我妻子Holly本身也是个儿童读物作家，所以孩子们已经习惯了我俩关起房门赶工期。但他们仍然自始至终都快乐地鼓励我们。Holly将这一切转化成了动力。她从来没跟我提起过在我撰写第3版的时候，她总共从头到尾写完了多少本书，对此我只能默默点赞。

稍后我会列出正式的审稿人名单，在此我要特别感谢那些订购了第3版预览版的朋友们，他们指出了我的笔误并提出了修改建议，而且不断地询问什么时候能够正式出版。有一批热切期望早日看到本书出版的读者，对我来说是莫大的鼓舞。

我与Manning的团队相处得十分融洽，不管是因第1版的出版而熟识的朋友还是新加入者，与他们一同工作都非常愉快。关于内容的增删，Mike Stephens和Jeff Bleiel给出了极具参考性的意见，他们能把所有事都安排妥当。Andy Carroll和Katie Tennant分别进行了专业的编辑和校对，而且从来不曾对我的英式作风、吹毛求疵或一些常见困惑表现出任何的不耐烦。出版部门一如既往地在背后出色地工作着，无论如何，我要对他们表示感谢：Dottie Marsico、Janet Vail、Marija Tudor和Mary Piergies。最后，我要感谢出版人Marjan Bace，他使我得以撰写本书第3版，并且还就未来我们可能会采取的合作方式提出了一些有趣的建议。

同行评审也十分重要，这不仅可以规范技术细节，还能平衡并协调书中内容。有时我们得到的建议仅涉及书的整体架构，而有时则是关于极为具体的细节（这时我都作了相应的修改）。无论哪种形式的反馈，我都非常欢迎。因此要感谢以下审稿人，他们让本书的质量更上一层楼：Andy Kirsch、Bas Pennings、Bret Colloff、Charles M. Gross、Dror Helper、Dustin Laine、Ivan Todorovic、Jon Parish、Sebastian Martín Aguilar、Tiaan Geldenhuys和Timo Bredenoort。

特别感谢Stephen Toub和Stephen Cleary，他们对本书第15章的早期评审非常宝贵。异步是一个特别复杂的主题，要想写得既清楚又准确十分不易。他们的专业建议使这一章的质量有了极大提升。

① 在大多数版本控制工具中，绿色都代表新增的内容。作者借此来指代新增的两章内容。——译者注

当然，如果没有C#团队，本书根本不可能存在。他们在语言的设计、实现和测试方面的贡献无与伦比，我期待他们接下来提出的内容。第2版出版之后，Eric Lippert就离开了C#团队，开始了新的传奇之旅[1]。令我万分感激的是，他仍然愿意做第3版的技术评审。还要感谢他为本书第1版作序，这一版仍然沿用了它。本书自始至终都有提到他的独到见解，如果你没有读过他的博客文章（http://ericlippert.com），那么你真应该读一读。

① Eric Lippert的博客名称即为代码传奇之旅（Fabulous Adventure in Coding）。——译者注

关于本书

这是一本关于C# 2及后续版本的书——就是这么简单。关于C# 1我几乎没讲什么。至于.NET Framework库和CLR（公共语言运行时），我也只是讨论了它们同语言有关的那一部分。我故意如此，结果就是这本书显然有别于市面上的大多数C#和.NET书籍。

假定读者已经具备了一定的C# 1相关知识，我就不必再花几百页的篇幅来讲述大多数人都已经知道的知识。这样就可以有更多的篇幅来深挖C# 2、C# 3和C# 4的细节，这正是我希望你阅读本书的目的。我写本书第1版时，有些读者对C# 2还相当陌生。现在，几乎所有的C#开发人员用过了C# 2中引入的特性，但这一版仍然保留了对C# 2的介绍，因为它是后续内容的基础。

目标读者

本书面向对C#有所了解的开发人员。最起码要十分了解C# 1，对后续版本略知一二即可。能达到这个要求的读者已经不多了，但我相信仍有很多开发者会通过深入分析C# 2和C# 3而受益，即使他们已经用了一段时间。而且，有很多开发人员根本没有用过C# 4和C# 5。

如果你对C#一无所知，那本书可能不适合你阅读。你可以通过查找你不熟悉的部分来苦读本书，但这并不是有效的学习方法。你最好选另外一本书，然后循序渐进地将本书加入到你的书单中。有很多从头介绍C#的图书，这些书的风格千差万别。*C# in a nutshell*系列（O'Reilly）一直都是这方面的好书，*Essential C# 5.0*（Addison-Wesley Professional）也是不错的选择。

我不是说阅读本书你就会变成一位出色的编码者（coder）。除了知道你正用到的语法，在软件工程中还有许多东西要学。我会在书中适时提供一些指导，但在开发过程中，我们不得不承认，在很多地方，基本上都是要依赖于某种直觉的。我想说的是，如果你阅读并理解了这本书，那么在使用C#时会感觉更加顺手，而且会更加自然地凭自己的直觉行事，而不会有太多的犹豫。这不是说因为用到语言的一些“生僻”的功能，你写出来的代码别人便看不懂了。相反，我是说你将拥有强大的自信，知道自己有哪些选择，并知道C#语言本身在鼓励你选择哪条路走下去。

路线图

本书的结构很简单，共有5个部分和3个附录。第一部分相当于简介，包括对C# 1重点主题的回顾，这些主题对于理解C#的后续版本非常重要，而且经常被人误解。第二部分讲述了C# 2的新特性。第三部分讲述了C# 3的新特性，等等。

某些时候，像这样组织全书内容，意味着一个主题会被反复讲到——尤其是委托在C# 2中有所增强，在C# 3中则进一步增强。但是，我的疯狂中却蕴藏着深意。我估计许多读者在不同的项目中会使用不同版本的C#：例如，在工作中使用C# 4研发，而回到家则使用C# 5实验。所以，我觉得有必要明确哪个版本中有哪些内容。这样还可以营造出一种“现场感”和“进化感”——可以体会到随着时间的推移，语言是如何进化的。

第1章的作用是搭建起一个舞台，从一段简单的C# 1代码开始，让它不断演变，观察更高的版本如何使代码变得越来越易读，越来越强大。我们介绍了C#成长的历史背景，以及它作为一个完整平台的一部分而工作的技术背景。具体地说，C#作为一种语言，它的基础是各种各样的“框架库”（.NET Framework中的各种库）以及一个强大的运行时（runtime）。借助它们，我们可以将抽象的东西转变成现实。

第2章回顾了C# 1的3个特定方面：委托、类型系统的特征以及值类型和引用类型的差异。许多C# 1开发者对这些主题有了“足够”的认识，但由于C#对它们进行了极大的拓展，所以要充分利用那些新特性，就需要一个坚实的基础。

第3章探讨了C# 2最大、同时也有可能最难掌握的一个特性：泛型。方法和类型可以用泛型的方式来写，调用代码中指定的真实类型将被泛型定义中的“类型参数”代替。刚开始，这个说法确实会令人迷惑，但理解了泛型后，就会感觉自己再也离不开它们了。

如果你想过怎样表示一个空整数，第4章就是为你准备的。这一章介绍了可空类型，这是基于泛型建立起来的一个特性，利用了语言、运行时和框架所提供的支持。

第5章介绍了C# 2对委托的增强。在此之前，你也许只用委托处理过像按钮单击这样的事件。C# 2使委托更容易创建，而库的支持使它们在除了事件之外的其他场合更加有用。

第6章讨论了迭代器，以及在C# 2中实现它们的简易方式。很少有开发者使用迭代器块，但由于LINQ to Objects是建立在迭代器基础上的，所以它们会变得越来越重要。它们执行时的“惰性” 也是LINQ的一个关键部分。

第7章对C# 2引入的许多较小的特性进行了描述，它们使我们的编程工作变得更加令人愉悦。语言的设计者对C# 1的一些粗糙的设计进行了改进，现在能够更灵活地与代码生成器交互，能更好地支持工具类，能更细致地访问属性，等等。

第8章同样研究了一些相对简单的特性，但这一次是针对C# 3的。几乎所有新语法都针对LINQ这个共同的目标进行了调整。但是这些基本的构建单元本身也相当有用。使用匿名类型、自动实现的属性、隐式类型的局部变量以及得到显著增强的初始化支持，C# 3变成一个内容更加丰富的语言，使你的代码可以更好地表达程序的行为。

第9章探讨了C# 3的第一个重要主题——Lambda表达式。语言的设计者并不满足于第5章介绍得已经相当精简的语法，而是将委托变得比在C# 2中还要容易创建。Lambda表达式还能做更多的事情——它们可以转换成表达式树：这是以数据形式来表示代码的一种强大的方式。

第10章探讨了扩展方法，它提供了一种完美的方式来欺骗编译器，使编译器相信在一个类型中声明的方法实际上从属于另一个类型。从表面上看，这似乎会带来可读性上的灾难，但经过仔细考虑之后，你就会发现它是一个相当强大的特性——而且对LINQ来说至关重要。

第11章以查询表达式的形式合并了前面3章的内容。查询表达式是一种简单而又强大的数据查询方式。我们先是将重点放在LINQ to Objects上，但所谓举一反三，通过观察如何应用查询表达式模式，你就明白LINQ to Objects能轻松地替换成其他数据提供器。

第12章简要介绍了LINQ的不同用法。首先，我们学习了LINQ to SQL如何将看似普通的C#转换为SQL语句，以此来展示查询表达式和表达式树相结合的好处。然后，以LINQ to XML为例，继续介绍了如何将库设计得能够与LINQ相契合。Parallel LINQ（并行LINQ）和Reactive Extensions（响应式扩展）展示了进程内查询的两种替代方法。本章最后讨论了如何用自己的LINQ操作符扩展LINQ to Objects。

从第13章开始介绍C# 4，包括命名实参和可选参数、COM互操作的改进和泛型可变性。在某种程度上，这些特性彼此互不相干，但COM互操作以及用于处理COM对象的其他功能，都受益于命名实参和可选参数。

第14章介绍了C# 4中最重要的特性：动态类型。用执行时的动态成员绑定代替编译时的静态绑定，对C#来说是一个巨大的尝试。但它在应用时是有选择性的：只有那些与动态值相关的代码才会动态地执行。

第15章介绍的是异步。C# 5只包含一个主要特性——编写异步函数的能力。这个特性非常复杂，不能一下子掌握，但掌握以后，使用起来非常优雅。

在接近尾声的第16章介绍了C# 5的其他特性（两个小特性），又展望了一下未来。

附录列出了所有的参考资料。附录A通过一些示例，介绍了LINQ标准查询操作符。附录B展示了核心的泛型集合类和接口。附录C简要介绍了.NET的不同版本，包括精简框架和Silverlight。

术语、排版约定和下载

本书大多数术语都会在出现时予以解释，但有的定义值得在这里重点讲一讲。我会相当明确地使用C# 1、C# 2、C# 3、C# 4和C# 5，但在其他书籍和网站中，你也许会看到C# 1.0、C# 2.0、C# 3.0、C# 4.0和C# 5.0这样的写法。我认为“.0”是多余的，因此省略了它们，希望意思清晰明确。

我从Mark Michaelis的一本C#书中借用了两个术语。runtime这个词有两个意思，一个是执行环境（如公共语言运行时），另一个是某个时间点（如“覆盖在运行时发生”）。为避免混淆，Mark换用“执行时”表示后一种概念，通常与“编译时”对应。这对我来说是一个很好的思路，我希望在社区中推广这种区分方法。本书从我做起，沿用了Mark的思路。

我经常会简单地说“语言规范”或者只是“规范”，除特别说明外，它们表示的是“C#语言规范”。但是，规范实际上有多个版本，一部分原因是由于语言本身有不同的版本，另一部分原因是标准化的过程如此。本书提到的规范的节序号都来自微软的《C# 5.0语言规范》。

本书有大量代码片段，它们是用等宽字体印刷的。代码清单的输出同样如此。有的代码清单添加了旁注，而且有时一些特定的代码段会加粗，以突出内容的变化、改进或增加。几乎所有代码都以代码段的形式出现，目的是保持精简，同时仍然可以在合适的环境中运行。那个环境就是

Snippy，这是1.8节将介绍的一个定制工具。Snippy可以从csharpindepth.com站点下载，本书所有代码也可以从此站点下载，有的是代码段形式，有的是完整的Visual Studio解决方案的形式，更多时候这两种形式都有。也可以从Manning出版社的网站manning.com/CSharpinDepthThirdEdition上下载。

Author Online和本书网站

购买本书，你可以访问由Manning出版社运行的一个内部论坛，可以在这里发表关于本书的评论，询问技术问题，并得到作者和其他用户的帮助。为了访问论坛并注册，请打开www.manning.com/CSharpinDepth ThirdEdition。网页上介绍了注册后进入论坛的方法，可以获得什么帮助，以及论坛的规则是什么。

只要书一出版，Author Online论坛和以前的讨论内容就可以从出版社的网站访问。

除了Manning出版社网站，我还为本书建立了一个配套网站，网址是www.csharpindepth.com，上面有许多不适合放到书中的内容和本书所有代码清单以及其他资源的链接。

目　　录

第一部分　基础知识

第 1 章　C#开发的进化史 ……2

1.1　从简单的数据类型开始 ……3
1.1.1　C# 1 中定义的产品类型 ……3
1.1.2　C# 2 中的强类型集合 ……4
1.1.3　C# 3 中自动实现的属性 ……5
1.1.4　C# 4 中的命名实参 ……6
1.2　排序和过滤 ……7
1.2.1　按名称对产品进行排序 ……7
1.2.2　查询集合 ……10
1.3　处理未知数据 ……12
1.3.1　表示未知的价格 ……12
1.3.2　可选参数和默认值 ……13
1.4　LINQ 简介 ……14
1.4.1　查询表达式和进程内查询 ……14
1.4.2　查询 XML ……15
1.4.3　LINQ to SQL ……16
1.5　COM 和动态类型 ……17
1.5.1　简化 COM 互操作 ……17
1.5.2　与动态语言互操作 ……18
1.6　轻松编写异步代码 ……19
1.7　剖析.NET 平台 ……20
1.7.1　C#语言 ……20
1.7.2　运行时 ……21
1.7.3　框架库 ……21
1.8　怎样写出超炫的代码 ……22
1.8.1　采用代码段形式的全能代码 ……22
1.8.2　教学代码不是产品代码 ……23
1.8.3　你的新朋友：语言规范 ……23
1.9　小结 ……24

第 2 章　C# 1 所搭建的核心基础 ……25

2.1　委托 ……25
2.1.1　简单委托的构成 ……26
2.1.2　合并和删除委托 ……30
2.1.3　对事件的简单讨论 ……32
2.1.4　委托总结 ……33
2.2　类型系统的特征 ……33
2.2.1　C#在类型系统世界中的位置 ……34
2.2.2　C# 1 的类型系统何时不够用 ……36
2.2.3　类型系统特征总结 ……39
2.3　值类型和引用类型 ……39
2.3.1　现实世界中的值和引用 ……39
2.3.2　值类型和引用类型基础知识 ……40
2.3.3　走出误区 ……41
2.3.4　装箱和拆箱 ……43
2.3.5　值类型和引用类型小结 ……44
2.4　C# 1 之外：构建于坚实基础之上的新特性 ……44
2.4.1　与委托有关的特性 ……44
2.4.2　与类型系统有关的特性 ……46
2.4.3　与值类型有关的特性 ……48
2.5　小结 ……49

第二部分　C# 2：解决C# 1 的问题

第 3 章　用泛型实现参数化类型 ……52

3.1　为什么需要泛型 ……53
3.2　日常使用的简单泛型 ……54
3.2.1　通过例子来学习：泛型字典 ……54
3.2.2　泛型类型和类型参数 ……56
3.2.3　泛型方法和判读泛型声明 ……59

3.3 深化与提高 62
3.3.1 类型约束 62
3.3.2 泛型方法类型实参的类型推断 67
3.3.3 实现泛型 68
3.4 高级泛型 73
3.4.1 静态字段和静态构造函数 73
3.4.2 JIT 编译器如何处理泛型 75
3.4.3 泛型迭代 77
3.4.4 反射和泛型 79
3.5 泛型在 C#和其他语言中的限制 82
3.5.1 泛型可变性的缺乏 83
3.5.2 缺乏操作符约束或者“数值”约束 87
3.5.3 缺乏泛型属性、索引器和其他成员类型 88
3.5.4 同 C++模板的对比 89
3.5.5 和 Java 泛型的对比 90
3.6 小结 91

第 4 章 可空类型 93

4.1 没有值时怎么办 93
4.1.1 为什么值类型的变量不能是 null 94
4.1.2 在 C# 1 中表示空值的模式 94
4.2 System.Nullable<T>和 System.Nullable 96
4.2.1 Nullable<T>简介 96
4.2.2 Nullable<T>装箱和拆箱 99
4.2.3 Nullable<T>实例的相等性 100
4.2.4 来自非泛型 Nullable 类的支持 101
4.3 C# 2 为可空类型提供的语法糖 101
4.3.1 ?修饰符 102
4.3.2 使用 null 进行赋值和比较 103
4.3.3 可空转换和操作符 105
4.3.4 可空逻辑 108
4.3.5 对可空类型使用 as 操作符 109
4.3.6 空合并操作符 110
4.4 可空类型的新奇用法 112
4.4.1 尝试一个不使用输出参数的操作 113
4.4.2 空合并操作符让比较不再痛苦 115
4.5 小结 117

第 5 章 进入快速通道的委托 118

5.1 向笨拙的委托语法说拜拜 119
5.2 方法组转换 120
5.3 协变性和逆变性 122
5.3.1 委托参数的逆变性 122
5.3.2 委托返回类型的协变性 123
5.3.3 不兼容的风险 124
5.4 使用匿名方法的内联委托操作 125
5.4.1 从简单的开始：处理一个参数 126
5.4.2 匿名方法的返回值 128
5.4.3 忽略委托参数 129
5.5 匿名方法中的捕获变量 131
5.5.1 定义闭包和不同类型的变量 131
5.5.2 捕获变量的行为 132
5.5.3 捕获变量到底有什么用处 133
5.5.4 捕获变量的延长生存期 134
5.5.5 局部变量实例化 135
5.5.6 共享和非共享的变量混合使用 137
5.5.7 捕获变量的使用规则和小结 139
5.6 小结 140

第 6 章 实现迭代器的捷径 141

6.1 C# 1：手写迭代器的痛苦 142
6.2 C# 2：利用 yield 语句简化迭代器 144
6.2.1 迭代器块和 yield return 简介 145
6.2.2 观察迭代器的工作流程 146
6.2.3 进一步了解迭代器执行流程 148
6.2.4 具体实现中的奇特之处 151
6.3 真实的迭代器示例 152
6.3.1 迭代时刻表中的日期 152
6.3.2 迭代文件中的行 153
6.3.3 使用迭代器块和谓词对项进行延迟过滤 156
6.4 使用 CCR 实现伪同步代码 157
6.5 小结 160

第 7 章 结束 C# 2 的讲解：最后的一些特性……161
7.1 分部类型……162
7.1.1 在多个文件中创建一个类型……162
7.1.2 分部类型的使用……164
7.1.3 C# 3 独有的分部方法……166
7.2 静态类型……167
7.3 独立的取值方法/赋值方法属性访问器……169
7.4 命名空间别名……170
7.4.1 限定的命名空间别名……171
7.4.2 全局命名空间别名……172
7.4.3 外部别名……173
7.5 pragma 指令……174
7.5.1 警告 pragma……174
7.5.2 校验和 pragma……175
7.6 非安全代码中固定大小的缓冲区……176
7.7 把内部成员暴露给选定的程序集……178
7.7.1 简单情况下的友元程序集……178
7.7.2 为什么使用 InternalsVisibleTo……179
7.7.3 InternalsVisibleTo 和签名程序集……179
7.8 小结……180

第三部分 C# 3：革新写代码的方式

第 8 章 用智能的编译器来防错……182
8.1 自动实现的属性……183
8.2 隐式类型的局部变量……185
8.2.1 用 var 声明局部变量……185
8.2.2 隐式类型的限制……187
8.2.3 隐式类型的优缺点……188
8.2.4 建议……189
8.3 简化的初始化……190
8.3.1 定义示例类型……190
8.3.2 设置简单属性……191
8.3.3 为嵌入对象设置属性……192
8.3.4 集合初始化程序……193
8.3.5 初始化特性的应用……196
8.4 隐式类型的数组……197
8.5 匿名类型……198
8.5.1 第一次邂逅匿名类型……198
8.5.2 匿名类型的成员……200
8.5.3 投影初始化程序……201
8.5.4 重点何在……202
8.6 小结……203

第 9 章 Lambda 表达式和表达式树……204
9.1 作为委托的 Lambda 表达式……205
9.1.1 准备工作：Func<...>委托类型简介……205
9.1.2 第一次转换成 Lambda 表达式……206
9.1.3 用单一表达式作为主体……207
9.1.4 隐式类型的参数列表……207
9.1.5 单一参数的快捷语法……208
9.2 使用 List<T>和事件的简单例子……209
9.2.1 列表的过滤、排序和操作……210
9.2.2 在事件处理程序中进行记录……211
9.3 表达式树……212
9.3.1 以编程方式构建表达式树……213
9.3.2 将表达式树编译成委托……214
9.3.3 将 C# Lambda 表达式转换成表达式树……215
9.3.4 位于 LINQ 核心的表达式树……218
9.3.5 LINQ 之外的表达式树……220
9.4 类型推断和重载决策的改变……221
9.4.1 改变的起因：精简泛型方法调用……221
9.4.2 推断匿名函数的返回类型……222
9.4.3 分两个阶段进行的类型推断……223
9.4.4 选择正确的被重载的方法……227
9.4.5 类型推断和重载决策……229
9.5 小结……229

第 10 章 扩展方法……230
10.1 未引入扩展方法之前的状态……231
10.2 扩展方法的语法……233
10.2.1 声明扩展方法……233
10.2.2 调用扩展方法……234
10.2.3 扩展方法是怎样被发现的……235

10.2.4 在空引用上调用方法 236
10.3 .NET 3.5 中的扩展方法 238
10.3.1 从 Enumerable 开始起步 238
10.3.2 用 Where 过滤并将方法调用链接到一起 240
10.3.3 插曲：似曾相识的 Where 方法 241
10.3.4 用 Select 方法和匿名类型进行投影 242
10.3.5 用 OrderBy 方法进行排序 243
10.3.6 涉及链接的实际例子 244
10.4 使用思路和原则 245
10.4.1 “扩展世界”和使接口更丰富 246
10.4.2 流畅接口 246
10.4.3 理智使用扩展方法 248
10.5 小结 249

第 11 章 查询表达式和 LINQ to Objects 250
11.1 LINQ 介绍 251
11.1.1 LINQ 中的基础概念 251
11.1.2 定义示例数据模型 255
11.2 简单的开始：选择元素 256
11.2.1 以数据源作为开始，以选择作为结束 257
11.2.2 编译器转译是查询表达式基础的转译 257
11.2.3 范围变量和重要的投影 260
11.2.4 Cast、OfType 和显式类型的范围变量 262
11.3 对序列进行过滤和排序 264
11.3.1 使用 where 子句进行过滤 264
11.3.2 退化的查询表达式 265
11.3.3 使用 orderby 子句进行排序 265
11.4 let 子句和透明标识符 267
11.4.1 用 let 来进行中间计算 267
11.4.2 透明标识符 268
11.5 连接 270
11.5.1 使用 join 子句的内连接 270
11.5.2 使用 join...into 子句进行分组连接 274
11.5.3 使用多个 from 子句进行交叉连接和合并序列 276
11.6 分组和延续 279
11.6.1 使用 group...by 子句进行分组 279
11.6.2 查询延续 282
11.7 在查询表达式和点标记之间作出选择 285
11.7.1 需要使用点标记的操作 285
11.7.2 使用点标记可能会更简单的查询表达式 286
11.7.3 选择查询表达式 286
11.8 小结 287

第 12 章 超越集合的 LINQ 289
12.1 使用 LINQ to SQL 查询数据库 290
12.1.1 数据库和模型 290
12.1.2 用查询表达式访问数据库 292
12.1.3 包含连接的查询 294
12.2 用 IQueryable 和 IQueryProvider 进行转换 296
12.2.1 IQueryable<T>和相关接口的介绍 297
12.2.2 模拟接口实现来记录调用 298
12.2.3 把表达式粘合在一起：Queryable 的扩展方法 300
12.2.4 模拟实际运行的查询提供器 302
12.2.5 包装 IQueryable 303
12.3 LINQ 友好的 API 和 LINQ to XML 303
12.3.1 LINQ to XML 中的核心类型 304
12.3.2 声明式构造 305
12.3.3 查询单个节点 308
12.3.4 合并查询操作符 309
12.3.5 与 LINQ 和谐共处 310
12.4 用并行 LINQ 代替 LINQ to Objects 311

12.4.1 在单线程中绘制曼德博罗特集 ······311
12.4.2 ParallelEnumerable、ParallelQuery 和 AsParallel ······313
12.4.3 调整并行查询 ······315
12.5 使用 LINQ to Rx 反转查询模型 ······316
12.5.1 IObservable<T>和 IObserver<T> ······316
12.5.2 简单的开始 ······318
12.5.3 查询可观察对象 ······319
12.5.4 意义何在 ······321
12.6 扩展 LINQ to Objects ······321
12.6.1 设计和实现指南 ······322
12.6.2 示例扩展：选择随机元素 ······323
12.7 小结 ······324

第四部分　C# 4：良好的交互性

第 13 章　简化代码的微小修改 ······328
13.1 可选参数和命名实参 ······328
13.1.1 可选参数 ······329
13.1.2 命名实参 ······334
13.1.3 两者相结合 ······337
13.2 改善 COM 互操作性 ······341
13.2.1 在 C# 4 之前操纵 Word 是十分恐怖的 ······342
13.2.2 可选参数和命名实参的复仇 ······342
13.2.3 按值传递 ref 参数 ······343
13.2.4 调用命名索引器 ······344
13.2.5 链接主互操作程序集 ······345
13.3 接口和委托的泛型可变性 ······348
13.3.1 可变性的种类：协变性和逆变性 ······348
13.3.2 在接口中使用可变性 ······349
13.3.3 在委托中使用可变性 ······352
13.3.4 复杂情况 ······353
13.3.5 限制和说明 ······354
13.4 对锁和字段风格的事件的微小改变 ······357
13.4.1 健壮的锁 ······357
13.4.2 字段风格的事件 ······358
13.5 小结 ······359

第 14 章　静态语言中的动态绑定 ······360
14.1 何谓、何时、为何、如何 ······361
14.1.1 何谓动态类型 ······361
14.1.2 动态类型什么时候有用，为什么 ······362
14.1.3 C# 4 如何提供动态类型 ······363
14.2 关于动态的快速指南 ······364
14.3 动态类型示例 ······366
14.3.1 COM 和 Office ······367
14.3.2 动态语言 ······368
14.3.3 纯托管代码中的动态类型 ······372
14.4 幕后原理 ······377
14.4.1 DLR 简介 ······378
14.4.2 DLR 核心概念 ······379
14.4.3 C#编译器如何处理动态 ······382
14.4.4 更加智能的 C#编译器 ······385
14.4.5 动态代码的约束 ······388
14.5 实现动态行为 ······390
14.5.1 使用 ExpandoObject ······391
14.5.2 使用 DynamicObject ······394
14.5.3 实现 IDynamicMetaObjectProvider ······400
14.6 小结 ······404

第五部分　C# 5：简化的异步编程

第 15 章　使用 async/await 进行异步编程 ······406
15.1 异步函数简介 ······407
15.1.1 初识异步类型 ······408
15.1.2 分解第一个示例 ······409
15.2 思考异步编程 ······410
15.2.1 异步执行的基础 ······410
15.2.2 异步方法 ······412
15.3 语法和语义 ······413
15.3.1 声明异步方法 ······413

15.3.2 异步方法的返回类型 ············ 414
15.3.3 可等待模式 ············ 415
15.3.4 await 表达式的流 ············ 418
15.3.5 从异步方法返回 ············ 421
15.3.6 异常 ············ 422
15.4 异步匿名函数 ············ 429
15.5 实现细节：编译器转换 ············ 431
15.5.1 生成的代码 ············ 432
15.5.2 骨架方法的结构 ············ 434
15.5.3 状态机的结构 ············ 435
15.5.4 一个入口搞定一切 ············ 436
15.5.5 围绕 await 表达式的控制 ············ 438
15.5.6 跟踪栈 ············ 439
15.5.7 更多内容 ············ 440
15.6 高效地使用 async/await ············ 441
15.6.1 基于任务的异步模式 ············ 441
15.6.2 组合异步操作 ············ 444
15.6.3 对异步代码编写单元测试 ············ 447
15.6.4 可等待模式的归来 ············ 450
15.6.5 在 WinRT 中执行异步操作 ············ 451
15.7 小结 ············ 452

第 16 章 C# 5 附加特性和结束语 ············ 453

16.1 foreach 循环中捕获变量的变化 ············ 453
16.2 调用者信息特性 ············ 454
16.2.1 基本行为 ············ 454
16.2.2 日志 ············ 456
16.2.3 实现 INotifyProperty-Changed ············ 456
16.2.4 在非.NET 4.5 环境下使用调用者信息特性 ············ 457
16.3 结束语 ············ 458

附录 A LINQ 标准查询操作符 ············ 460

附录 B .NET 中的泛型集合 ············ 471

附录 C 版本总结 ············ 483

Part 1 第一部分

基础知识

每位读者都对本书有一些不同的期待，而且不同读者的经验和技能水平不同。你可能是一个专家，希望填补现有知识体系的一些小的空白；或者你将自己视为一个“普通”的开发者，对泛型和Lambda表达式有一点了解，渴望能够更好地运用它们；又或者你能够得心应手地使用C# 2和C# 3进行开发，却对C# 4或C# 5毫无经验。

作为作者，我无法将每个读者都变得一模一样——即使能，也不愿意。然而，我希望所有读者都有两点共性：渴望更深入了解C#这门语言；至少对C# 1有基本的掌握。如果你满足这两点，剩下的事情就可以交给我了。

由于读者的技能水平可能相差较大，本部分内容非常必要。你可能已经知道应该从更高版本的C#那里期待一些什么，也可能对它们完全陌生。你可能已经非常牢靠地掌握了C# 1，或者对一些技术细节还感到生疏，这些细节在以前或许不太重要，但在C#后续版本中，却显得越来越重要。等到第一部分结束时，虽然不能保证每个人都在完全相同的起跑线上，但你会更有信心学习本书剩余部分，对后面的内容也能做到心中有数。

在前两章中，我们既会“向前看”，也会“向后看”。本书的关键主题之一就是进化（或者说演变）。在将任何特性引入语言之前，设计团队都会把这个特性放到现有的环境中，同时还会就其总体发展目标，对其进行严格的考察。这样一来，语言就获得了一种让人感觉非常舒服的“一致性”，即使是在语言逐渐发生变革的过程中，它的一致性也得到了良好的保持。为了理解语言如何进化以及为什么进化，我们既要回顾一下历史，也要明确一下未来的发展方向。

第1章对全书内容进行了概述，简单讨论了C# 1之后版本的一些最重要的特性，并演示了代码从C# 1到现在的演变过程。由于新特性不断地应用于编程，最初粗陋的代码已了无痕迹，我们目前使用的代码越来越臻于完善。我还将介绍本书所使用的术语，以及示例代码的格式。

第2章将重点放在C# 1上。如果你已经是一名C# 1专家，可以跳过这一章。不过，它确实解释清楚了C# 1中一些很容易让人“犯迷糊”的地方。我们不会尝试对整个语言进行解释，而是将重点放在作为C#后续版本基础的一些特性上。在此基础上，才能顺利进入第二部分对C# 2的讨论。

第 1 章

C#开发的进化史

1

本章内容

- ❑ 一个进化的例子
- ❑ .NET的组成
- ❑ 使用本书代码
- ❑ C#语言规范

你知道我喜欢Python、Ruby、Groovy这些动态语言的哪些方面吗？它们去代码之糟粕，仅取其精华，即真正进行处理的那部分内容。繁琐的形式被生成器、Lambda表达式、列表推导式等特性所代替。

有趣的是，那些旨在让动态语言感觉轻巧的特性，却很少与动态有关。当然，像鸭子类型和活动记录（Active Record）中的一些神奇用法还是与动态有关的，但静态类型的语言却完全可以不那么笨拙和繁重。

回到C#。在某些方面，C# 1可以看做2001年Java语言的升级版。它们的相似之处十分明显，但C#还包含一些额外的特性：作为一级语言特性的属性、委托和事件、`foreach`循环、`using`语句、显式方法重载、操作符重载、自定义值类型等，恕不一一列举。语言的偏好显然是个人问题，但我第一次使用C# 1时，就明显感到它比Java要更进一步。

从那时起，C#的开发可谓渐入佳境。C#的每个新版本都添加了重要的特性，不断为开发者排忧解难，而这些特性往往都是经过深思熟虑的，很少有向后不兼容的情况。即使在C# 4增加真正实用的动态类型功能之前，C#也已经引入了很多与动态（及函数式）语言相关的特性，使代码更加容易编写和维护。类似地，围绕C# 5中异步的功能与F#中的这些功能还不完全相同，但我觉得已经有了进步。

本书将逐一介绍这些变化，让你真正喜欢上C#编译器准备为你上演的奇迹。不过这些都是后话，本章我将马不停蹄地介绍尽可能多的特性。在比较C#语言和.NET平台时，我会给出我的定义，随后还要重点说明一下本书剩余部分应掌握的关键内容。接着我们就可以深入细节了。

我们不会在这单独的一章中介绍所有变化，但会涵盖以下方面的内容：泛型、设置不同访问修饰符的属性、可空类型、匿名方法、自动实现的属性、增强的集合初始化程序、增强的对象初始化程序、Lambda表达式、扩展方法、隐式类型、LINQ查询表达式、命名实参、可选参数、简

化的COM互操作和动态类型。这些将伴随着我们从C# 1一路讲到最新发布的C# 5。内容显然很多，现在就开始吧。

1.1 从简单的数据类型开始

本章，我将让C#编译器实现一些神奇的功能，但不会告诉你怎么做，也很少会提及这是什么以及为什么会这样。这是唯一一次我不会解释原理，或者说不会按部就班地展示例子的情形。事实上，我的计划是先给你留下一个不可磨灭的印象，而不是教给你具体的知识。在你读完本节的内容之后，如果对C#能做的事情仍然没有感到丝毫兴奋，那么本书或许真的不适合你。相反，如果你迫切地想知道我的“魔术”是怎么玩的——想让我放慢“手法”，直至看清楚发生的所有事情——那么这正是本书剩余部分要做的事情。

事先要提醒你的是，这个例子可能显得十分牵强——是为了在尽可能短的代码中包含尽可能多的新特性而“生搬硬造”的。此外，这个例子还有点“老生常谈”，但至少你会觉得它比较眼熟。这里展示的是一个产品/名称/价格（product/name/price）的例子，是“hello, world”程序的电子商务版。我们将看到各种任务是如何实现的，体验随着C#版本的提高，如何更简单、更优雅地完成相同的任务。C# 5的优势放到最后介绍，但不要担心，这样并不会影响它的重要性。

1.1.1 C# 1 中定义的产品类型

我们将以定义一个表示产品的类型作为开始，然后进行处理。在`Product`类型内部，没有特别吸引人的东西，它只是封装了几个属性。为方便演示，我们还要在这个地方创建预定义产品的一个列表。

代码清单1-1展示了用C# 1写的`Product`类型。稍后，还会演示在更高版本中如何达到相同的效果。我们将按照这个模式演示其他所有代码。由于这些代码是在2013年编写的，因此你可能已经熟悉了将要介绍的某些特性，不过回头看看还是值得的，我们可以看看这门语言到底有了多少进步。

代码清单1-1 `Product`类型（C# 1）

```
using System.Collections;
public class Product
{
   string name;
   public string Name { get { return name; } }

   decimal price;
   public decimal Price { get { return price; } }

   public Product(string name, decimal price)
   {
      this.name = name;
      this.price = price;
   }
```

```
    public static ArrayList GetSampleProducts()
    {
        ArrayList list = new ArrayList();
        list.Add(new Product("West Side Story", 9.99m));
        list.Add(new Product("Assassins", 14.99m));
        list.Add(new Product("Frogs", 13.99m));
        list.Add(new Product("Sweeney Todd", 10.99m));
        return list;
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", name, price);
    }
}
```

代码清单1-1没有什么难以理解的东西——它毕竟只是C# 1代码。然而，它确实证明了C# 1代码存在如下3个局限。

- ArrayList没有提供与其内部内容有关的编译时信息。不慎在GetSampleProducts创建的列表中添加一个字符串是完全有可能的，而编译器对此没有任何反应。
- 代码中为属性提供了公共的取值方法，这意味着如果添加对应的赋值方法，那么赋值方法也必须是公共的。
- 用于创建属性和变量的代码很复杂——封装一个字符串和一个十进制数应该是一个十分简单的任务，不该这么复杂。

来看看C# 2作了哪些改进。

1.1.2　C# 2 中的强类型集合

我们所做的第一组改动（如代码清单1-2所示）针对上面列出的前两项，包含C# 2中最重要的改变：泛型。新的内容用粗体列出。

代码清单1-2　强类型集合和私有的赋值方法（C# 2）

```
public class Product
{
    string name;
    public string Name
    {
        get { return name; }
        private set { name = value; }
    }

    decimal price;
    public decimal Price
    {
        get { return price; }
        private set { price = value; }
    }

    public Product(string name, decimal price)
```

```
    {
        Name = name;
        Price = price;
    }

    public static List<Product> GetSampleProducts()
    {
        List<Product> list = new List<Product>();
        list.Add(new Product("West Side Story", 9.99m));
        list.Add(new Product("Assassins", 14.99m));
        list.Add(new Product("Frogs", 13.99m));
        list.Add(new Product("Sweeney Todd", 10.99m));
        return list;
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", name, price);
    }
}
```

现在，属性拥有了私有的赋值方法（我们在构造函数中使用了这两个赋值方法）。并且它能非常“聪明”地猜出`List<Product>`是告知编译器列表中只能包含`Product`。试图将一个不同的类型添加到列表中，会造成编译时错误，并且当你从列表中获取结果时，也并不需要转换结果的类型。

C# 2解决了原先的3个问题中的2个。代码清单1-3展示了C# 3如何解决剩下的那个问题。

1.1.3 C# 3 中自动实现的属性

我们先从C# 3中相对乏味的一些特性开始。代码清单1-3中自动实现的属性和简化的初始化，相比Lambda表达式等特性来说，有点微不足道，不过它们可以大大地简化代码。

代码清单1-3 自动实现的属性和更简单的初始化（C# 3）

```
using System.Collections.Generic;

class Product
{
    public string Name { get; private set; }
    public decimal Price { get; private set; }

    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }

    Product() {}

    public static List<Product> GetSampleProducts()
    {
        return new List<Product>
        {
```

```
            new Product { Name="West Side Story", Price = 9.99m },
            new Product { Name="Assassins", Price=14.99m },
            new Product { Name="Frogs", Price=13.99m },
            new Product { Name="Sweeney Todd", Price=10.99m}
        };
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", Name, Price);
    }
}
```

现在，不再有任何代码（或者可见的变量）与属性关联，而且硬编码的列表是以一种全然不同的方式构建的。由于没有name和price变量可供访问，我们必须在类中处处使用属性，这增强了一致性。现在有一个私有的无参构造函数，用于新的基于属性的初始化。（设置这些属性之前，会对每一项调用这个构造函数。）

在本例中，实际上可以完全删除旧的公共构造函数。但这样一来，外部代码就不能再创建其他的产品实例了。

1.1.4　C# 4 中的命名实参

对于C# 4，涉及属性和构造函数时，我们需要回到原始代码。其中有一个原因是为了让它不易变：尽管拥有私有赋值方法的类型不能被公共地改变，但如果它也不能被私有地改变[①]，将会显得更加清晰。不幸的是，对于只读属性，没有快捷方式。但C# 4允许我们在调用构造函数时指定实参的名称，如代码清单1-4所示，它为我们提供了和C# 3的初始化程序一样的清晰度，而且还移除了易变性（mutability）。

代码清单1-4　命名实参带来了清晰的初始化代码（C# 4）

```
using System.Collections.Generic;
public class Product
{
    readonly string name;
    public string Name { get { return name; } }

    readonly decimal price;
    public decimal Price { get { return price; } }

    public Product(string name, decimal price)
    {
        this.name = name;
        this.price = price;
    }

    public static List<Product> GetSampleProducts()
    {
        return new List<Product>
```

① 面向C# 1的那段代码本来也是不可变的，让它可变是为了简化面向C# 2和C# 3的代码段的修改。

```
        {
            new Product( name: "West Side Story", price: 9.99m),
            new Product( name: "Assassins", price: 14.99m),
            new Product( name: "Frogs", price: 13.99m),
            new Product( name: "Sweeney Todd", price: 10.99m)
        };
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", name, price);
    }
}
```

在这个特定的示例中，该特性的好处不是很明显，但当方法或构造函数包含多个参数时，它可以使代码的含义更加清楚——特别是当参数类型相同，或某个参数为null时。当然，你可以选择什么时候使用该特性，只在使代码更好理解时才指定参数的名称。

图1-1总结了Product类型的演变历程。完成了每个任务之后，我都会提供一幅类似的示意图，便于你体会C#在增强代码时，遵循的是一个什么样的模式。注意，图中没有涉及C# 5。这是因为C# 5主要特性（异步函数）面向的领域还没有太多的语言支持。稍后，我们将简单看一下。

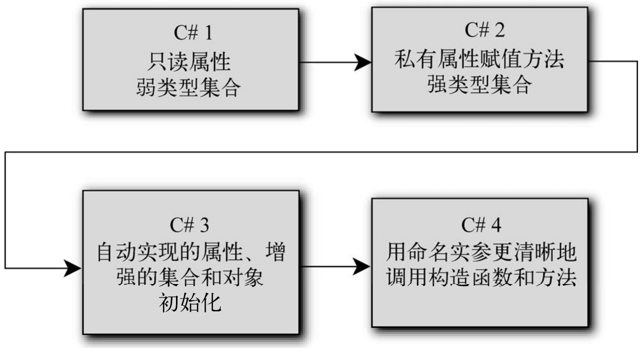


图1-1　Product类型的演变历程，展示了越来越好的封装性、越来越强的类型化以及越来越容易的初始化

到目前为止，你看到的变化幅度都不大。事实上，泛型的加入（List<Product>语法）或许是C# 2最重要的一个部分。但是，我们现在只看到了它的部分用处。虽然没有什么让人心跳加快的东西，但我们只是刚刚开始。下一个任务是以字母顺序打印产品列表。

1.2　排序和过滤

本节不会改变Product类型，我们会使用示例的产品列表，并按名称排序，然后找出最贵的产品。每个任务都不难，但我们可以看到它到底能简化到什么程度。

1.2.1　按名称对产品进行排序

以特定顺序显示一个列表的最简单方式就是先将列表排好序，再遍历并显示其中的项。

在.NET 1.1中，这要求使用ArrayList.Sort，而且在我们的例子中，要求提供一个IComparer实现。也可以让Product类型实现IComparable，但那就只能定义一种排序顺序。很容易就会想到，以后除了需要按名称排序，还可能需要按价格排序。

代码清单1-5实现了IComparer，然后对列表进行排序，并显示它。

代码清单1-5　使用IComparer对ArrayList进行排序（C# 1）

```
class ProductNameComparer : IComparer
{
   public int Compare(object x, object y)
   {
      Product first = (Product)x;
      Product second = (Product)y;
      return first.Name.CompareTo(second.Name);
   }
}
...
ArrayList products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
   Console.WriteLine (product);
}
```

在代码清单1-5中，要注意的第一件事是，必须引入一个额外的类型来帮助排序。虽然这并不是一个大问题，但假如在一个地方只是想按名称进行排序，就会感觉编码工作过于繁重。其次，注意Compare方法中的强制类型转换。强制类型转换相当于告诉编译器："嘿嘿，我知道的比你多一点点。"但是，这也意味着你可能是错误的。如果从GetSampleProducts返回的ArrayList包含一个字符串，那么代码会出错——因为在比较时试图将字符串强制转型为Product。

在给出排序列表的代码中也进行了强制类型转换。这个转换不如刚才的转换明显，因为是编译器自动进行的。foreach循环会隐式将列表中的每个元素转换为Product类型。同样，这种情况在执行时会失败，在C# 2中，"泛型"可以帮助我们解决这些问题。在代码清单1-6中，唯一的改变就是引入了泛型。

代码清单1-6　使用IComparer<Product>对List<Product>进行排序（C# 2）

```
class ProductNameComparer : IComparer<Product>
{
   public int Compare(Product x, Product y)
   {
      return x.Name.CompareTo(y.Name);
   }
}
...
List<Product> products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
   Console.WriteLine(product);
}
```

在代码清单1-6中，对产品名进行比较的代码变得更简单，因为一开始提供的就是Product（而不可能是其他类型）。不需要进行强制类型转换。类似地，foreach循环中隐式的类型转换也被取消了。编译器仍然会考虑将序列中的源类型转换为变量的目标类型，但它知道这时两种类型均为Product，因此没必要产生任何用于转换的代码。

确实有了一定的改进。但是，我们希望能直接指定要进行的比较，就能开始对产品进行排序，而不需要实现一个接口来做这件事。代码清单1-7展示了具体如何做，它告诉Sort方法如何用一个委托来比较两个产品。

代码清单1-7 使用Comparison<Product>对List<Product>进行排序（C# 2）

```
List<Product> products = Product.GetSampleProducts();

products.Sort(delegate(Product x, Product y)
   { return x.Name.CompareTo(y.Name); }
);
foreach (Product product in products)
{
   Console.WriteLine(product);
}
```

注意，现在已经不再需要ProductNameComparer类型了。以粗体印刷的语句实际会创建一个委托实例。我们将这个委托提供给Sort方法来执行比较。第5章会更多地讲解这个特性（*匿名方法*）。

现在，我们已经修正了在C# 1的版本中不喜欢的所有东西。但是，这并不是说C# 3不能做得更好。首先，将匿名方法替换成一种更简洁的创建委托实例的方式，如代码清单1-8所示。

代码清单1-8 在Lambda表达式中使用Comparison<Product>进行排序（C# 3）

```
List<Product> products = Product.GetSampleProducts();
products.Sort((x, y) => x.Name.CompareTo(y.Name));
foreach (Product product in products)
{
   Console.WriteLine(product);
}
```

你又看到了一种奇怪的语法（一个*Lambda表达式*），它仍然会像代码清单1-7那样创建一个Comparison<Product>委托，只是代码量减少了。这里不必使用delegate关键字来引入委托，甚至不需要指定参数类型。

除此之外，使用C# 3还有其他好处。现在，可以轻松地按顺序打印名称，同时不必修改原始产品列表。代码清单1-9使用OrderBy方法对此进行了演示。

代码清单1-9 使用一个扩展方法对List<Product>进行排序（C# 3）

```
List<Product> products = Product.GetSampleProducts();
foreach (Product product in products.OrderBy(p => p.Name) )
{
   Console.WriteLine (product);
}
```

这里似乎调用了一个`OrderBy`方法，但查阅一下MSDN，就会发现这个方法在`List<Product>`中根本不存在。之所以能调用它，是由于存在一个*扩展方法*，第10章将讨论扩展方法的细节。这里实际不再是“原地”对列表进行排序，而只是按特定的顺序获取列表的内容。有时，你需要更改实际的列表；但有时，没有任何副作用的排序显得更“善解人意”。

重点在于，现在的写法更简洁，可读性更好（当然是在你理解了语法之后）。我们的想法是“列表按名称排序”，现在的代码正是这样做的。并不是“列表通过将一个产品的名称与另一个产品的名称进行比较来排序”，就像C# 2代码所做的那样。也不是使用知道如何将一个产品与另一个产品进行比较的另一种类型的实例来排序。这种简化的表达方式是C# 3的核心优势之一。既然单独的数据查询和操作是如此简单，那么在执行更大规模的数据处理时，仍然可以保持代码的简洁性和可读性，这进而鼓励开发者以一种“以数据为中心”的方式来观察世界。

本节又展示了一小部分C# 2和C# 3的强大功能，还有许多尚待解释的语法。但是，即使你还不理解这背后的细节，趋势也是相当明朗的：我们正在向更清晰、更简单的代码迈进！图1-2展示了这个演变过程。

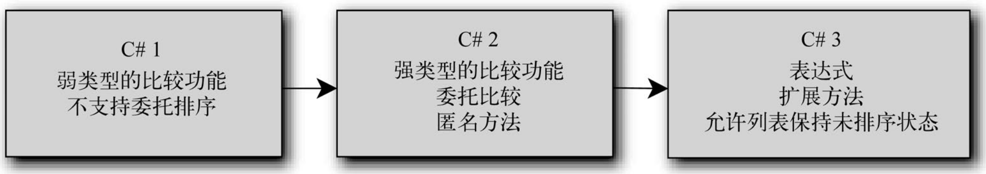


图1-2　在C# 2和C# 3中用于简化排序的特性

到目前为止，我们只讲了排序[1]。现在来讨论一种不同的数据处理方式——查询。

1.2.2　查询集合

下一个任务是找出列表中符合特定条件的所有元素。具体地说，要找出价格高于10美元的产品。在C# 1中，需要运行循环，测试每个元素，并打印出符合条件的元素（参见代码清单1-10）。

代码清单1-10　循环、测试和打印（C# 1）

```
ArrayList products = Product.GetSampleProducts();
foreach (Product product in products)
{
   if (product.Price > 10m)
   {
      Console.WriteLine(product);
   }
}
```

好吧，上面的代码写起来不难，也很容易理解。然而，请注意3个任务是如何交织在一起的：

[1] C# 4也提供了一个跟排序有关的特性，叫做*泛型可变性*（generic variance），这里如果举例的话需要较长篇幅。第13章末尾有详细介绍。

用foreach进行循环，用if测试条件，再用Console.WriteLine显示产品。这3个任务的依赖性是一目了然的，看看它们是如何嵌套的就明白了。

C# 2稍微进行了一下改进（参见代码清单1-11）。

代码清单1-11　测试和打印分开进行（C# 2）

```
List<Product> products = Product.GetSampleProducts();

Predicate<Product> test = delegate(Product p) { return p.Price > 10m; };
List<Product> matches = products.FindAll(test);

Action<Product> print = Console.WriteLine;
matches.ForEach(print);
```

变量test的初始化使用了上节介绍的匿名方法，而print变量的初始化使用了C# 2的另一个特性——*方法组转换*，它简化了从现有方法创建委托的过程。

我不是说上述代码要比C# 1的代码简单，只是说它要强大[①]得多。

具体地说，它使我们可以非常轻松地更改测试条件并对每个匹配项采取单独的操作。涉及的委托变量（test和print）可以传递给一个方法——相同的方法可以用于测试完全不同的条件以及执行完全不同的操作。当然，可以将所有测试和打印都放到一条语句中，如代码清单1-12所示。

代码清单1-12　测试和打印分开进行的另一个版本（C# 2）

```
List<Product> products = Product.GetSampleProducts();
products.FindAll(delegate(Product p) { return p.Price > 10;})
        .ForEach(Console.WriteLine);
```

这样更好一些，但delegate(Product p)还是很碍事，大括号也是。它们是代码中的不和谐音符，有损可读性。如果一直进行相同的测试和执行相同的操作，我还是喜欢C# 1的版本。（虽然说起来很平常，但还是要提醒你，完全可以在使用C# 2或C# 3时使用C# 1的版本。谁都不会用推土机来种植郁金香，我们这里使用的技术显得有点儿“小题大做”了。）

C# 3拿掉了以前将实际的委托*逻辑*包裹起来的许多无意义的东西，从而有了极大的改进（参见代码清单1-13）。

代码清单1-13　用Lambda表达式来测试（C# 3）

```
List<Product> products = Product.GetSampleProducts();
foreach (Product product in products.Where(p => p.Price > 10))
{
   Console.WriteLine(product);
}
```

Lambda表达式将测试放在一个非常恰当的位置。再加上一个有意义的方法名，你甚至能大声念出代码，几乎不用怎么思考就能理解代码的含义。C# 2的灵活性也得到了保留——传递给Where的参数值可以来源于一个变量。此外，如果愿意，完全可以使用Action<Product>，而不是硬编码的Console.WriteLine调用。

① 从某种角度看，这个说法有点儿言过其实。完全可以在C# 1中定义恰当的委托，并在循环中调用它们。.NET 2.0的FindAll和ForEach方法只是鼓励你多分解问题。

本节的这个任务强调了我们通过前面的排序任务已经明确的一点——使用匿名方法可以轻松编写一个委托，Lambda表达式则更进一步，将这个任务变得更简单。换言之，可以在foreach循环的第一个部分中包含查询或排序操作，同时不会影响代码的可读性。

图1-3总结了这些编程方式上的变化。对于这个任务来说，C# 4没有提供任何可以进一步简化的特性。

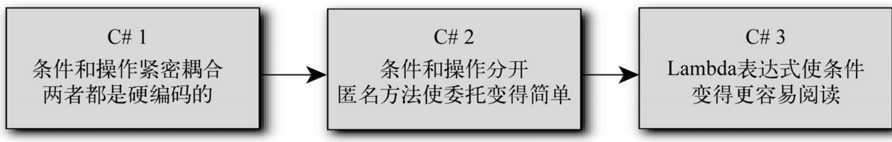


图1-3　在C# 2中，匿名方法有助于问题的可分离性；在C# 3中，Lambda表达式则增强了可读性

现在，我们已经给出了过滤过的列表，接下来假设我们的数据跟以前不一样了。如果并非总是知道一个产品的价格，那么会发生什么？如何在Product类中应对这个问题？

1.3　处理未知数据

我们将要介绍两种不同形式的未知数据。首先，处理确实没有数据信息的场景。其次，再来看看如何从方法调用中移除信息，使用默认值来代替。

1.3.1　表示未知的价格

这一次不打算展示太多的代码，但问题肯定是你熟悉的，尤其是假如你经常使用数据库的话。假定产品列表不仅包含现售的产品，还包括尚未面市的产品。某些情况下，我们可能不知道价格。如果decimal是引用类型，那么只需使用null来表示未知的价格。但是，由于它是值类型，我们不能这样表示。那么，在C# 1中如何表示？有3种常见的解决方案：

- 围绕decimal创建一个引用类型包装器；
- 维护一个单独的Boolean标志，它表示价格是否已知；
- 使用一个“魔数”（magic value）（比如decimal.MinValue）来表示未知价格。

你得承认，其中没有一个方案是特别好的。神奇的是，在变量和属性声明中添加一个额外的字符，即可解决这个问题。.NET 2.0通过引入Nullable<T>结构，C# 2通过提供一些语法糖（syntactic sugar），使事情得到了极大的简化。现在可以将属性声明更改为如下代码块：

```
decimal? price;
public decimal? Price
{
   get { return price; }
   private set { price = value; }
}
```

构造函数的参数也更改为decimal?。这样一来，就可以将null作为参数值传递进来，或者在类中写Price=null;。null的含义从“不指向任何对象的一个特殊引用”变成“代表没有给出其他数据的任意可空类型的一个特殊值”，其中所有引用类型和基于Nullable<T>的类型被视为可空类型。

这比其他任何解决方案都更有表现力。代码的其余部分和往常一样工作——价格未知的产品默认价格低于10美元，因为可空值是通过“大于”操作符来处理比较的①。为了检查一个价格是否已知，可以把它同null比较，或者使用HasValue属性。所以，为了在C# 3中显示所有价格未知的产品，可以像代码清单1-14这样写。

代码清单1-14 显示价格未知的产品（C# 3）

```
List<Product> products = Product.GetSampleProducts();
foreach (Product product in products.Where(p => p.Price == null))
{
   Console.WriteLine(product.Name);
}
```

C# 2代码与代码清单1-12相似，但使用了return p.Price == null;作为匿名方法的方法体。

```
List<Product> products = Product.GetSampleProducts();
products.FindAll(delegate(Product p) { return p.Price == null; })
        .ForEach(Console.WriteLine);
```

C# 3在可空类型方面没有进行什么改进，而C# 4则提供了一个与之相关的特性。

1.3.2 可选参数和默认值

有时你并不想给出方法所需的所有东西，比如对于某个特定参数，你可能总是会使用同样的值。传统的解决方案是对该方法进行重载，现在C# 4引入的*可选参数*（optional parameter）可以简化这一操作。

在Product类型的C# 4版本中，构造函数接收产品的名称和价格。在C# 2和C# 3中，我们可以将价格设置为可空的decimal类型，但现在我们假设大多数产品都不包含价格。如果能像下面这样初始化产品就再好不过了：

```
Product p = new Product("Unreleased product");
```

在C# 4之前，我们只能添加一个Product构造函数的重载来实现这一目的。而使用C# 4可以为价格参数声明一个默认值（在本例中为null）：

```
public Product(string name, decimal? price = null)
{
```

① 在作者的网站上，对这句话进行了更清楚的说明，原文翻译如下：原来我在书中的那句话之所以讲得不清楚，是由于用大于操作符来执行和$10的比较。与null值进行大小比较，结果始终是false。所以，假如在比较的时候不是写成price > 10m，而是写成看起来应该相同的!(price <= 10m)，就会得到错误的答案。这恰好印证了4.3.2节在讲述“涉及可空类型的操作符”主题时所讲的话。——译者注

```
        this.name = name;
        this.price = price;
    }
```

你需要为声明的可选参数指定一个常量值。这个值不一定为null，只不过在本例中默认值恰好为空而已。它可以应用于任何类型的参数，但是对于除字符串之外的引用类型来说，只能使用null作为可用的常量值。

图1-4总结了C#不同版本的演变。

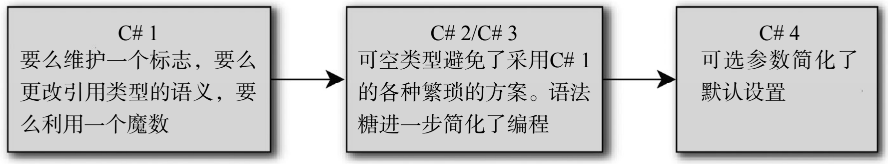


图1-4　处理“未知”数据的方法

到现在为止，所有的特性都很有用，但也许都不值得大书特书。下面我们来看一个让人更加兴奋的特性：LINQ。

1.4　LINQ 简介

LINQ（Language Integrated Query，语言集成查询），是C# 3的核心。顾名思义，LINQ是关于查询的，其目的是使用一致的语法和特性，以一种易阅读、可组合的方式，使对多数据源的查询变得简单。

在很大程度上，C# 2更像是对C# 1的各种不足之处的修修补补，所以并没有一鸣惊人。而C# 3中几乎所有特性都是为了构建LINQ，并且其结果也十分特别。我见过其他语言中的一些特性，可以解决一些与LINQ相同领域的问题，但没有一个可以如此全面和灵活。

1.4.1　查询表达式和进程内查询

如果你以前见过LINQ，肯定会知道*查询表达式*可以以一种声明式风格对不同数据源创建查询。之所以前面的示例都没有使用查询表达式，是因为那些例子*不*使用查询表达式反而更简单。当然，这并不是说不能使用。例如，代码清单1-15与代码清单1-13是等价的。

代码清单1-15　使用查询表达式的前几步：过滤集合

```
List<Product> products = Product.GetSampleProducts();
var filtered = from Product p in products
               where p.Price > 10
               select p;
foreach (Product product in filtered)
{
   Console.WriteLine(product);
}
```

我个人认为早先的代码清单（代码清单1-13）更易读——查询表达式唯一的好处就是where子句显得更简单。这里我还偷偷使用了另一个特性——*隐式类型局部变量*（implicitly typed local variable），它使用var上下文关键字声明。编译器可以根据该变量的初始值推断其类型。因此，filtered的类型为IEnumerable<Product>。本章后面的示例中将大量使用var。它对于写书来说也是十分重要的，可以节省代码清单的空间。

那么，如果查询表达式不好，为什么每个人都对它（和LINQ）如此看重呢？第一个答案是，虽然查询表达式不是特别适合简单任务，但在一些较复杂的情况下，如果换成用方法调用来写（尤其是用C# 1或C# 2的方式），代码会变得难以阅读。在这些情况下，查询表达式就显得非常好用。为了稍微增大一点难度，引入另一个类型——Supplier（供货商）。

每个供货商都有一个Name(string)和一个SupplierID(int)。我在Product类添加了SupplierID属性，并对示例数据进行了适当的改编。无可否认，为每个产品都提供一个供货商，这不是一种纯面向对象的方式——但它更接近于数据在数据库中的表示方式。目前，这样做更易于演示这个特性（查询表达式）。但到第12章时，你会看到LINQ也允许我们使用一个更自然的模型。

现在来看一下代码（代码清单1-16）。它将示例产品与示例供货商连接起来（明显要基于供货商的ID来连接），并对产品使用和之前一样的价格过滤器，先按供货商名排序，再按产品名排序，最后打印每个匹配项的供货商名称和产品名称。这要放在以前版本的C#中，不知道需要输入多少代码，而且实现起来简直就是一场噩梦。相反，在LINQ中，要做到这些事情实在是太容易了。

代码清单1-16 连接（joining）、过滤（filtering）、排序（ordering）和投影（projecting）（C# 3）

```
List<Product> products = Product.GetSampleProducts();
List<Supplier> suppliers = Supplier.GetSampleSuppliers();
var filtered = from p in products
               join s in suppliers
                  on p.SupplierID equals s.SupplierID
               where p.Price > 10
               orderby s.Name, p.Name
               select new { SupplierName = s.Name, ProductName = p.Name };
foreach (var v in filtered)
{
   Console.WriteLine("Supplier={0}; Product={1}",
                     v.SupplierName, v.ProductName);
}
```

明眼人一看便知，这跟SQL实在太像了。事实上，许多人初识LINQ时（但在仔细研究它之前），第一个反应就是抗拒，因为它似乎纯粹就是将SQL引入了语言，目的是为了加强与数据库交互的能力。幸好，LINQ只是借用了SQL的语法和一些思路。正如我们见到的那样，不需要数据库就能使用它，到现在为止，我所运行过的代码中，没有任何代码与数据库有关。事实上，可以从任意来源（如XML）获取数据。

1.4.2 查询XML

假定不是将供货商和产品硬编码进来，而是使用以下XML文件：

```
<?xml version="1.0"?>
<Data>
  <Products>
    <Product Name="West Side Story" Price="9.99" SupplierID="1" />
    <Product Name="Assassins" Price="14.99" SupplierID="2" />
    <Product Name="Frogs" Price="13.99" SupplierID="1" />
    <Product Name="Sweeney Todd" Price="10.99" SupplierID="3" />
  </Products>

  <Suppliers>
    <Supplier Name="Solely Sondheim" SupplierID="1" />
    <Supplier Name="CD-by-CD-by-Sondheim" SupplierID="2" />
    <Supplier Name="Barbershop CDs" SupplierID="3" />
  </Suppliers>
</Data>
```

虽然这个文件非常简单,但从中提取数据的最佳方式是什么?怎样查询它?怎样基于它来进行连接操作?肯定会比代码清单1-16难吧?代码清单1-17展示了在LINQ to XML中要做多少工作。

代码清单1-17　用LINQ to XML对XML文件进行“复杂”的处理(C# 3)

```
XDocument doc = XDocument.Load("data.xml");
var filtered = from p in doc.Descendants("Product")
               join s in doc.Descendants("Supplier")
                  on (int)p.Attribute("SupplierID")
                  equals (int)s.Attribute("SupplierID")
               where (decimal)p.Attribute("Price") > 10
               orderby (string)s.Attribute("Name"),
                       (string)p.Attribute("Name")
               select new
              {
                   SupplierName = (string)s.Attribute("Name"),
                   ProductName = (string)p.Attribute("Name")
              };
foreach (var v in filtered)
{
    Console.WriteLine("Supplier={0}; Product={1}",
                      v.SupplierName, v.ProductName);
}
```

我得承认，现在的代码不像前面那么直观了，因为需要告诉系统如何理解数据(什么属性应该作为什么类型使用)。但是，两者的差别并不太大。尤其是在两个代码清单的每个部分之间，存在着明显的联系。假如不是因为行长的限制需要断行，在两个查询之间，应该是逐行对应的。

印象深刻吗?还没有完全信服吗?我们将数据放到一个它更有可能存在的地方——数据库中。

1.4.3　LINQ to SQL

此时要做一些工作(大多数都是自动进行的)让LINQ to SQL知道什么数据表里该有什么内容，但整个过程还是相当直观的。如代码清单1-18所示，我们将直接跳到查询代码。如果你想看`LinqDemoDataContext`的细节，在可下载的源代码中能找到。

代码清单1-18 对SQL数据库应用查询表达式（C# 3）

```
using (LinqDemoDataContext db = new LinqDemoDataContext())
{
var filtered = from p in db.Products
               join s in db.Suppliers
                  on p.SupplierID equals s.SupplierID
               where p.Price > 10
               orderby s.Name, p.Name
               select new { SupplierName = s.Name, ProductName = p.Name };
foreach (var v in filtered)
{
   Console.WriteLine("Supplier={0}; Product={1}",
                     v.SupplierName, v.ProductName);
   }
}
```

这些代码看起来就应该非常熟悉了。join那一行下面的所有内容都是直接从代码清单1-16中复制并粘贴过来的，没有进行任何改动。

这显然使人印象深刻，但思路清晰的人马上会想到一个问题：为什么要将所有数据都从数据库里“拽”回来，再应用这些.NET查询和排序呢？为什么不直接让数据库来做这些事情呢？那不正是它擅长的事情吗？事实上，这正是LINQ to SQL所做的事情。代码清单1-18中的代码发出了一个数据库请求，它基本上被转换为SQL查询。虽然查询是用C#代码来表示的，但却是作为SQL来执行的。

以后会知道，当模式（schema）和实体（entity）知道了供货商和产品之间的关系后，可以使用一种更面向关系（relation-oriented）的方式来进行连接。但结果是相同的。这里的例子只是展示了LINQ to Objects（对集合进行操作的“内存中的LINQ”）与LINQ to SQL是多么相似。

1.5 COM和动态类型

我要介绍的最后一个特性是C# 4特有的。LINQ是C# 3的主要内容，而互操作性是C# 4最重要的主题。这包括处理旧的COM技术，以及在DLR（Dynamic Language Runtime，动态语言运行时）上执行的全新动态语言。我们要将产品列表导出到一个Excel数据表中。

1.5.1 简化COM互操作

要让数据出现在Excel中有很多种方式，但使用COM来控制是最强大最灵活的。可惜，以前的C#操作起COM来实在是太复杂，VB就要好得多。C# 4扭转了这种情况。

代码清单1-19展示了如何将数据保存到新的电子表格中。

代码清单1-19 使用COM将数据保存到Excel中（C# 4）

```
var app = new Application { Visible = false };
Workbook workbook = app.Workbooks.Add();
Worksheet worksheet = app.ActiveSheet;
int row = 1;
foreach (var product in Product.GetSampleProducts()
```

```
                                .Where(p => p.Price != null))
    {
       worksheet.Cells[row, 1].Value = product.Name;
       worksheet.Cells[row, 2].Value = product.Price;
       row++;
    }
    workbook.SaveAs(Filename: "demo.xls",
                    FileFormat: XlFileFormat.xlWorkbookNormal);
    app.Application.Quit();
```

尽管这可能并没有我们想象得那么完美，但已经比C#以前的版本好多了。事实上，你已经在此了解了一些C# 4的特性，但这里有下面几个并不太明显的新特性。

- 使用命名实参调用SaveAs。
- 调用函数时，许多可选参数都可以省略实参。特别是SaveAs，正常情况下它可能还会有10个额外的实参！
- C# 4可以将PIA（Primary Interop Assembly，主互操作程序集）的相关部分内嵌到调用代码中，因此不必再单独部署PIA。
- 在C# 3中，由于ActiveSheet属性的类型为object，因此对worksheet赋值时如果不强制转换则会失败。在使用内嵌的PIA特性时，ActiveSheet的类型变为dynamic，从而使其他所有特性得以实现。

此外，在操作COM时，C# 4还支持命名索引器，本例没有演示该特性。

我已经透露了最后要介绍的特性：使用全新dynamic类型的动态类型。

1.5.2　与动态语言互操作

动态类型是一个非常大的主题，所以用整个第14章来介绍。这里我只介绍一个小小的示例，向你展示它能做什么。

假设我们的产品没有存储在数据库、XML或内存中。我们可以通过Web服务来访问，但只能使用Python代码。Web服务中的代码使用了Python的动态特性来构建结果，没有声明你要访问的属性的类型。相反，它要求你来指定属性，并试图在执行时理解你的意图。对于Python这类语言来说，这些都是很平常的事情。但如何用C#来访问结果呢？

答案是使用dynamic——一个新的类型[①]，C#编译器允许你动态地使用该类型。如果一个表达式为dynamic类型，你可以调用其方法、访问其属性、将其作为方法的参数进行传递，等等。并且大多数常见的绑定过程都发生在执行时，而不是编译时。你可以将dynamic类型的值隐式转换为其他类型（因此在代码清单1-19中可以对工作表进行那样的转换）或其他有趣的东西。

即使在纯粹的C#代码中，这种功能也很有用，虽然不需要COM互操作，但可能会需要与动态语言交互，而这时往往会更有用。代码清单1-20展示了如何从IronPython中获取产品列表并打印出来。它还包括一些设置代码，可以在同一进程中运行Python代码。

① 在某种程度上来说，是这样的。它对C#编译器来说是一个类型，但CLR却根本不认识它。

代码清单1-20 运行IronPython并动态获取属性（C# 4）

```
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.ExecuteFile("FindProducts.py");
dynamic products = scope.GetVariable("products");
foreach (dynamic product in products)
{
   Console.WriteLine("{0}: {1}", product.ProductName, product.Price);
}
```

products和product都声明为动态类型，因此编译器允许我们对产品列表进行迭代并打印其属性，尽管它并不知道这是否可以执行成功。如果不小心出现笔误，如将product.ProductName写成了product.Name，也只能在执行时才知道失败。

这与C#的其他部分截然相反，它们是静态类型的。动态类型只有在表达式为dynamic时才有效：大多数C#代码都会自始至终保持静态类型。

1.6 轻松编写异步代码

最终你会看到C# 5的超级特性：异步函数。我们可以用它来中断代码的执行，而不阻塞线程。

这个话题很大，非常大，但现在我只给出一个简单的代码片段。你肯定会意识到，Windows Forms中的线程有两条金科玉律：不能阻塞UI线程，并且不能在任何其他线程中访问UI元素（除非使用一些明确指定的方法）。代码清单1-21展示了Windows Forms应用程序中的一个处理按钮点击的方法，并根据产品ID显示产品信息。

代码清单1-21 使用异步函数在Windows Forms中显示产品

```
private async void CheckProduct(object sender, EventArgs e)
{
   try
   {
      productCheckButton.Enabled = false;
      string id = idInput.Text;

      Task<Product> productLookup = directory.LookupProductAsync(id);
      Task<int> stockLookup = warehouse.LookupStockLevelAsync(id);
      Product product = await productLookup;
      if (product == null)
      {
         return;
      }
      nameValue.Text = product.Name;
      priceValue.Text = product.Price.ToString("c");

      int stock = await stockLookup;
      stockValue.Text = stock.ToString();
      }
   finally
   {
   productCheckButton.Enabled = true;
   }
}
```

完整的方法要比代码清单1-21长，还包括在开头显示状态消息和清除结果的代码，但这个代码清单包含了所有重要的部分。我们加粗显示了新的语法——方法的async修饰符和两个await表达式。

即使忽略这些语法，你也能理解这段代码的大体流程。它先在产品目录和库存中查找产品详细信息和当前库存。然后等待，直到找到产品信息，如果目录中没有条目与给定的ID对应，就退出。否则，将产品名称和价格显示在UI元素上，然后再等待获得库存信息并显示。

产品和库存的查找都是异步的，可能操作数据库，也可能调用Web服务，这都不重要。在等待结果时，并没有真正阻塞UI线程，即使方法中的所有代码都运行于UI线程。结果返回时，线程从它离开的地方继续执行。该示例还演示了标准的流控制（try/finally）操作，完全跟你期望的一样。这段代码真正令人惊奇的地方在于，它正确地实现了你希望的异步操作，但却没有任何启动线程/BackgroundWorker、调用Control.BeginInvoke或对异步事件附加回调函数那样的冗繁代码。当然你仍然需要思考，异步并没有因async/await而变得容易，但却变得不再冗长。它在很大程度上去掉了模板代码，让你更加专注于你想控制的逻辑。

晕了吗？放松一些，本书其他部分会讲得慢得多。具体来说，我会解释一些个别情况并深入阐述引入不同特性的原因，另外还将就使用这些特性的时机给出一些指导。

现在，我已经向你展示了C#的特性。其中有一些同时是库的特性，有一些同时是运行时（runtime）的特性。我还会对此进行更详细的解释，现在先让我们弄清楚我所说的是什么。

1.7　剖析.NET 平台

最开始引入时，.NET这个词涵义甚广，用来包罗微软公司的多种技术。例如，Windows Live ID曾被叫做.NET Passport，虽然它和目前的.NET没有任何明显的联系。幸好，这个混乱的局面逐渐平息下来了。本节要探讨.NET的各个组成部分。

本书会提到3种不同的特性：C#语言本身的特性、运行时的特性（可以认为运行时提供了程序运行的一个“引擎”）以及.NET框架库的特性。本书重点是在C#语言上。通常只有在与C#本身的特性有关时，才会解释运行时和框架的特性。但是，只有在你清楚地理解了三者的差异之后，才能真正理解这些特性。特性经常会发生重叠，但重要的是理解其中的基本原理。

1.7.1　C#语言

C#语言是由它的规范定义的。C#规范描述了C#源代码的格式，其中包括语法和行为。规范中并没有描述编译器输出要在什么平台上运行，只描述了两者进行交互的要点。例如，C#语言需要一个名为System.IDisposable的类型，其中包含一个名为Dispose的方法。它们是定义using语句所必需的。同样，平台需要（不管以什么样的形式）同时支持值类型和引用类型，另外还要支持垃圾回收。

理论上，任何平台只要支持要求的特性，C#编译器就可以以它为目标平台。例如，C#编译器除了可以以IL（Intermediate Language，中间语言，是本书写作期间最常见的一种输出形式）

形式输出外，还可以以其他合法形式输出。运行时完全可以对C#编译器的输出进行解释，或将其直接完全转换为本地代码（native code），而不必非要对它进行JIT编译。尽管这些情况比较少见，但它们确实存在。例如，微框架[①]使用了解释器，Mono（http://xamarin.com/ios）也是如此。另一方面，NGen和Xamarin.iOS（构建iPhone和其他iOS应用的平台，参见http://monotouch.net/）都使用了前期编译（ahead-of-time compilation）。

1.7.2　运行时

.NET平台的运行时部分是数量相当少的一些代码，它们负责确保用IL写的程序以符合CLI（Common Language Infrastructure，公共语言基础设施）规范（ECMA-335和ISO/IEC 23271）Partitions I ~ III[②]的方式执行。CLI的运行时部分称为CLR（公共语言运行时）。本书以后提到CLR时，是指微软实现的CLR。

语言的一些元素永远不会在运行时的级别上出现，但也有一些元素越过了这个界线。例如，枚举器（enumerator）就不是在运行时的级别上定义的。相比之下，虽然数组和委托对`IDisposable`接口来说没有任何特别的含义，但它们对于"运行时"来说都是十分重要的。

1.7.3　框架库

库提供了可供我们在程序中使用的代码。.NET中的框架库主要是以IL的形式构建的，只有在必要时才使用本地代码。这是运行时优势的一个体现：你自己写的代码并非天生就是"二等公民"——它完全能够提供与它利用的库一样强大的功能和性能。库中的代码量要比运行时的代码量多得多，这跟车和引擎的关系是一样的。

.NET库得到了部分标准化。CLI规范的Partition IV提供了大量不同的概要（profile）协议和核心内容库。Partition IV包含两个部分。第一部分是对库的常规文字描述，包括哪些配置文件中包括哪些库。第二部分则以XML格式描述了库本身的细节。这是在C#中使用XML注释时生成的相同形式的文档。

不过，.NET中有许多东西都*不是*基本库定义的。如果写一个程序，在其中只使用来自规范定义的库，而且以正确的方式使用它们，那么代码应该能在任何实现（包括Mono、.NET等）上顺利地运行。但在实际应用中，几乎任意规模的任意程序都会使用非标准的库，如Windows Forms或ASP.NET。Mono项目也有它自己的、不是.NET一部分的库，如GTK#。另外，它也实现了许多非标准的库。

.NET一词是指微软公司提供的运行时和库的组合，其中也包含C#和VB.NET编译器。可以把它视为在Windows顶部构建的一个完整的开发平台。.NET各个部分的版本各不相同，这可能会造成混淆。附录C提供了一个概要，指明哪个部分的哪个版本在什么时候发布，以及包含哪些特性。

弄清楚这些之后，在开始深入研究C#之前，我还有一点需要强调。

---

① 详见附录C.5.3。——译者注

② CLI规范分为从I到VI的几个Partition。——译者注

1.8　怎样写出超炫的代码

很抱歉起了这么一个容易引起误会的标题。本节就此而言并不会让你的代码更优美。它甚至不会让你感觉清爽。但它对你理解本书大部分内容都会有所帮助，因此你一定要阅读本节。本来这种问题应该写在前面（第1页以前），但我知道很多读者都会跳过那部分，直接进入正文。我对此表示理解，因此我会尽快介绍这些内容。

1.8.1　采用代码段形式的全能代码

写一本有关计算机语言（脚本语言除外）的书时，其中的一个挑战就是完整的程序（无须添加额外的代码，读者能直接编译和运行的程序），这些程序很快就会变得非常长。我想解决这个问题，从而为你提供可以轻松键入和试验的代码：我认为实际键入代码的学习效果要比只读一读好得多。

只要有恰当的程序集引用和using指令，就能用极少量的C#代码做相当多的事情。但是，最麻烦的地方就是写那些using指令，然后声明一个类，然后声明一个Main方法。在所有这些“准备工作”都完成之后，才能动手写第一行真正有用的代码。我的例子基本上都是代码段的形式，忽略了在一个简单的程序中显得过于繁琐的那些“准备过程”，将精力集中在最重要的东西上面。我编写了一个小工具Snippy，可以直接运行这些代码段。

如果代码段不包含省略号（...），那么所有代码都将被认为是程序Main方法的方法体。如果有一个省略号，那么省略号之前的代码为方法和内嵌类的声明，之后的代码为Main方法的内容。例如下面的代码段：

```
static string Reverse(string input)
{
   char[] chars = input.ToCharArray();
   Array.Reverse(chars);
   return new string(chars);
}
...
Console.WriteLine(Reverse("dlrow olleH"));
```

Snippy可以将其扩展为下面的形式：

```
using System;
public class Snippet
{
    static string Reverse(string input)
    {
       char[] chars = input.ToCharArray();
       Array.Reverse(chars);
       return new string(chars);
    }

    [STAThread]
    static void Main()
```

```
        {
            Console.WriteLine(Reverse("dlrow olleH"));
        }
    }
```

实际上，Snippy所包含的using指令要多得多，但扩展后的版本已经很长了。注意，所生成的类永远都叫Snippet，在代码段中声明的任何类型都是该类的内嵌类。

本书的网站上详细介绍了如何使用Snippy（http://mng.bz/Lh82），还包含所有示例的代码段和位于Visual Studio解决方案中的扩展版本。此外，它还支持LINQPad（http://www.linqpad.net），它是由Joe Albahari开发的一个类似的工具，为研究LINQ提供了一些特别有用的功能。

接下来，我们来看一看之前的代码有没有什么不妥之处。

1.8.2　教学代码不是产品代码

如果你理解了本书所有代码，不进一步思考就直接将其用到你的应用中，这没有什么问题，但我强烈建议你不要这么做。大多数示例都是为了演示某个特定的知识点，仅此而已。例如，大多数代码段都不包含参数验证、访问修饰符、单元测试或文档。当这些代码段在超出其预期的上下文使用时，就可能会失败。

例如前面反转字符串的方法体，本书将多次使用这段代码。

```
char[] chars = input.ToCharArray();
Array.Reverse(chars);
return new string(chars);
```

先不管参数验证，这段代码可以反转字符串中以UTF-16编码的代码点序列，但在某些情况下，这还不够全面。比如一个由e和表示重音的组合字符组成的单个文字（即 é），不能交换它们在序列中的位置，否则重音将用在错误的字符上。再比如某个字符串包含从代理对形成的基本多文种平面（basic multilingual plane）之外的字符，重新排序可能会导致无效的UTF-16字符串。要修复这些问题需要更复杂的代码，反而容易让你忽略真正想表达的东西。

你可以随意使用本书中的代码，但请记住本节的忠告：从这些代码中获取灵感，这要比照抄照搬并认为它们能满足你特定的需求强多了。

最后，你还应该下载一本书，它涵盖了本书绝大部分内容。

1.8.3　你的新朋友：语言规范

我竭尽全力使本书正确无误，但是错误在所难免。你可以在本书的网站上查看已提交的错误列表（http://mng.bz/m1Hh）。如果你发现了任何错误，可以给我发邮件（skeet@pobox.com）或在作者论坛上发帖（http://mng.bz/TQmF）。不过，你有可能在收到我的反馈之前就已经解决了问题，又或许你的问题不在本书讨论范畴之内。归根结底，对于C#行为最权威的资源是语言规范。

规范有两种重要的形式——ECMA国标标准规范和微软规范。在撰写本书时，ECMA规范（ECMA-334和ISO/IEC 23270）尽管已是第4版，却只涵盖了C# 2。谁也不知道它是否会更新，以及什么时候更新，但微软的版本是完整的，并且是免费的。本书的网站上包含这两种规范所有可

用版本的链接（http://mng.bz/8s38），并且Visual Studio也自带了一份[1]。我在本书中所指的规范中的某一节，使用的是微软C# 4规范的序号，即便谈及的是之前的语言版本时也是如此。我强烈建议你下载该版本，并在遇到某些怪异的情况时及时查阅。

我的目标之一是让程序员人手一本规范，它提供一个更易于面向开发者的方式，覆盖日常编码中的方方面面，而不用记住编译器作者所要求的全部细节。话虽如此，作为一种规范它其实极其易读，你不应被吓到。如果你对规范感兴趣，有一个面向C# 3和C# 4带注解的版本。这两个版本都包含C#团队和其他贡献者添加的一些令人着迷的注释。（我声明：我是C# 4版本的一个贡献者，其他所有注释都非常棒！）

1.9　小结

本章展示了（但没有解释）本书要深入解析的一些特性。但还有许多没在这里展示，而且到现在为止见过的所有特性都有关联的“子特性”（subfeature）。希望本章的内容能激发你对本书剩余部分的兴趣。

本章大部分内容都是在介绍特性，不过我们也谈到了另外一些话题，从而帮助你从本书中获得最大收益。我解释了语言、运行时、库以及本书的代码形式。

讲解C# 2的特性之前，还有一个领域是必须涉及的，那就是C# 1。显然，作为一个作者，我不知道你对C# 1的了解程度如何。但是，我确实知道C# 1的哪些主题很难理解，有的主题是真正掌握更高版本C#的关键，所以下一章将详细讨论它们。

① 在不同的系统中，规范的具体位置也不同。但在Visual Studio 2012专业版中，它的位置是C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC#\Specifications\1033。

第2章 C# 1所搭建的核心基础

本章内容

- ❑ 委托
- ❑ 类型系统的特征
- ❑ 值/引用类型

本章不是对整个C# 1的全面复习。我也不会那样做。如果用一章的篇幅对C# 1进行全面讲述，那么我无法对每个主题做到“一视同仁”。我写这本书时假定读者至少对C# 1有一个基本的掌握。当然，多少才算“基本掌握”，这里面肯定存在一些主观因素。但是，我假定你至少是满怀信心地去应聘一份初级C#开发者的工作，能正确回答与工作相关的技术问题。我的期望是许多读者都有更多的经验，但这是我假设的知识水平。

本章的重点是3个C# 1主题，它们对理解更高版本的C#特性来说特别重要。这应该使“最小公分母”变得稍微大一点，使我能在本书以后进行稍大胆的假设 。假定这是“最小公分母”①，你可能会发现自己已经能很好地理解本章的所有概念。如果你认为你恰属于这种情况，那么完全可以跳过本章。如果以后发现有一些东西并不像你想得那么简单，那么随时可以回来。但是，或许至少应该看看每一节末尾的小结，那里罗列了重点——如果发现有自己不熟悉的东西，就应该详细阅读那一节。

我们先介绍委托，然后比较C#类型系统与其他语言的不同，最后介绍值类型和引用类型的区别。在各个主题中，我都将阐述概念和行为，并借此机会定义后面要用到的术语。在介绍完C# 1的工作原理之后，我们将快速浏览后续版本中有哪些与本章话题相关的新特性。

2.1 委托

可以肯定，你对委托（delegate）已经有了一个直观概念，只是无法清晰地说出来。如果你熟悉C语言，而且必须向另一个C程序员描述委托，你一定会立刻想到*函数指针*这个术语。实际

① 作者用“最小公分母”来形容所有读者都应该掌握的最起码的知识。掌握的基础知识越多，基础越牢靠，自然有利于以后理解更高深的主题。——译者注

上，委托在某种程度上提供了间接的方法。换言之，不需要直接指定一个要执行的行为，而是将这个行为用某种方式“包含”在一个对象中。这个对象可以像其他任何对象那样使用。在该对象中，可以执行封装的操作。可以选择将委托类型看做只定义了一个方法的接口，将委托的实例看做实现了那个接口的一个对象。

如果认为这些说法仍然很空洞，也许可以通过一个例子来加深理解。虽然这个例子有些离谱，但它的确能清楚地说明何为委托。让我们以遗嘱为例。遗嘱由一系列指令组成，比如：“付账单，捐善款，其余财产留给猫。”遗嘱一般是在某人去世之前写好，然后把它放到一个安全的地方。去世后，（希望）律师[①]会执行这些指令。

C#中的委托和现实世界的遗嘱一样，也是要在恰当的时间执行一系列操作。如果代码想要执行操作，但不知道操作细节，一般可以使用委托。例如，`Thread`类之所以知道要在一个新线程里运行什么，唯一的原因就是在启动新线程时，向它提供了一个`ThreadStart`或`ParameterizedThreadStart`委托实例。

为了开始我们的委托之旅，首先必须知道委托的4个基本条件，它们缺一不可。

2.1.1　简单委托的构成

为了让委托做某事，必须满足4个条件：

- ❑ 声明委托类型；
- ❑ 必须有一个方法包含了要执行的代码；
- ❑ 必须创建一个委托实例；
- ❑ 必须调用（invoke）委托实例。

下面依次讨论上述每一步。

1. 声明委托类型

委托类型实际上只是参数类型的一个列表以及一个返回类型。它规定了类型的实例能表示的操作。

例如，以如下方式声明一个委托类型：

```
delegate void StringProcessor(string input);
```

上述代码指出，如果要创建`StringProcessor`的一个实例，需要只带一个参数（一个字符串）的方法，而且这个方法要有一个`void`返回类型（该方法什么都不返回）。

这里的重点在于，`StringProcessor`其实是一个从`System.MulticastDelegate`派生的类型，后者又派生自`System.Delegate`。它有方法，可以创建它的实例，并将引用传递给实例，所有这些都没有问题。虽然它有一些自己的“特性”，但假如你对特定情况下发生的事情感到困惑，那么首先想一想使用“普通”的引用类型时发生的事情。

① 换言之，指示别人去做某事，但不知道他具体会怎么做，因为当事人已经死了。——译者注

说明　混乱的根源：容易产生歧义的“委托”　委托经常被人误解，这是由于大家喜欢用委托这个词来描述委托类型和委托实例。这两者的区别其实就是任何一个类型和该类型的实例的区别。例如，string类型本身和一组特定的字符肯定不同。委托类型和委托实例这两个词会贯穿本章始终，从而让你明白我具体说的是什么。

讨论委托的下一个基本元素时，会用到StringProcessor委托类型。

2. 为委托实例的操作找到一个恰当的方法

我们的下一个基本元素是找到（或者写）一个方法，它能做我们想做的事情，同时具有和委托类型相同的签名。基本的思路是，要确保在调用（invoke）一个委托实例时，使用的参数完全匹配，而且能以我们希望的方式（就像普通的方法调用）使用返回值（如果有的话）。

看看以下StringProcessor实例的5个备选方法签名：

```
void PrintString(string x)
void PrintInteger(int x)
void PrintTwoStrings(string x, string y)
int GetStringLength(string x)
void PrintObject(object x)
```

第1个方法完全符合要求，所以可以用它创建一个委托实例。第2个方法虽然也有一个参数，但不是string类型，所以不兼容StringProcessor。第3个方法第1个参数的类型匹配，但参数数量不匹配，所以也不兼容。第4个方法有正确的参数列表，但返回类型不是void。（如果委托类型有返回类型，方法的返回类型也必须与之匹配。）

第5个方法比较有趣，任何时候调用一个StringProcessor实例，都可以调用具有相同的参数的PrintObject方法，这是由于string是从object派生的。把这个方法作为StringProcessor的一个实例来使用是合情合理的，但C# 1要求委托必须具有完全相同的参数类型[①]。C# 2改善了这个状况——详见第5章。在某些方面，第4个方法也是相似的，因为总是可以忽略不需要的返回值。然而，void和非void返回类型目前一直被认为是不兼容的。部分原因是因为系统的其他方面（特别是JIT）需要知道，在执行方法时返回值是否会留在栈上[②]。

假定有一个针对兼容的签名（PrintString）的方法体。接着，讨论下一个基本元素——委托实例本身。

3. 创建委托实例

既然已经有了一个委托类型和一个有正确签名的方法，接着可以创建委托类型的一个实例，指定在调用委托实例时就执行该方法。虽然没有什么好的官方术语来定义这一行为，但在本书中，我会将该方法称为委托实例的操作。

至于具体用什么形式的表达式来创建委托实例，取决于操作使用实例方法还是静态方法。假

① 和参数的类型一样，参数的in（默认）out或ref前缀也必须匹配。然而，委托很少使用out/ref参数。

② 这里我故意含糊地使用了栈这个词，以避免太多不相关的细节。更多的信息请参阅Eric Lippert的博文The void is invariant（http://mng.bz/4g58）。

定PrintString是StaticMethods类型中的一个静态方法，在InstanceMethods类型中是一个实例方法。下面是创建一个StringProcessor实例的两个例子：

```
StringProcessor proc1, proc2;
proc1 = new StringProcessor(StaticMethods.PrintString);
InstanceMethods instance = new InstanceMethods();
proc2 = new StringProcessor(instance.PrintString);
```

如果操作是静态方法，指定类型名称就可以了。如果操作是实例方法，就需要先创建类型（或者它的派生类型）的一个实例。这和平时调用方法是一样的。这个对象[①]称为操作的目标。调用委托实例时，就会为这个对象调用[②]方法。如果操作在同一个类中（这种情况经常发生，尤其是在UI代码中写事件处理程序时），那么两种限定方式都不需要——实例方法[③]隐式将this引用作为前缀。同样，这些规则和你直接调用方法时没什么两样。

说明　最终的垃圾（或者不是，视情况而定）　必须注意，假如委托实例本身不能被回收，委托实例会阻止它的目标被作为垃圾回收。这可能造成明显的内存泄漏（leak），尤其是假如某“短命”对象调用了一个“长命”对象中的事件，并用它自身作为目标。“长命”对象间接容纳了对“短命”对象的一个引用，延长了“短命”对象的寿命。

单纯创建一个委托实例却不在某一时刻调用它是没有什么意义的。看看最后一步——调用。

4. 调用委托实例

这是很容易的一件事儿[④]，调用一个委托实例的方法就可以了。这个方法本身被称为Invoke。在委托类型中，这个方法以委托类型的形式出现，并且具有委托类型声明中指定的相同参数列表和返回类型。所以，在我们的例子中，有一个像下面这样的方法：

```
void Invoke(string input)
```

调用Invoke会执行委托实例的操作，向它传递在调用Invoke时指定的任何参数。另外，如果返回类型不是void，还要返回操作的返回值。

是不是很简单？C#将这个过程变得更简单——如果有一个委托类型的变量[⑤]，就可以把它视为方法本身。观察由不同时间发生的事件构成的一个事件链，很容易就可以理解这一点，如图2-1所示。

---

① 就是刚才创建的实例。——译者注

② “委托实例被调用”中的“调用”对应的是invoke，理解为“唤出”更恰当。它和后面的“为这个对象调用方法”中的“调用”稍有不同，后者对应的是call。在英语的语境中，invoke和call的区别在于，在执行一个所有信息都已知的方法时，用call比较恰当。这些信息包括要引用的类型、方法的签名以及方法名。但是，在需要先“唤出”某个东西来帮你调用一个信息不明的方法时，用invoke就比较恰当。但是，由于两者均翻译为“调用”不会对读者的理解造成太大的困扰，所以本书仍然采用约定俗成的方式来进行翻译。——译者注

③ 当然，如果操作是实例方法，并试图从静态方法中创建一个委托实例，就仍然需要提供一个引用作为目标。所谓“提供一个引用作为目标”是指仍然要写成proc2 = new StringProcessor(instance.PrintString);这样的形式，而不能写成proc2 = new StringProcessor(PrintString);。——译者注

④ 仅对同步调用而言。可以用BeginInvoke和EndInvoke来异步调用一个委托实例，但那超出了本章的范围。

⑤ 或其他任何种类的表达式，但通常是一个变量。

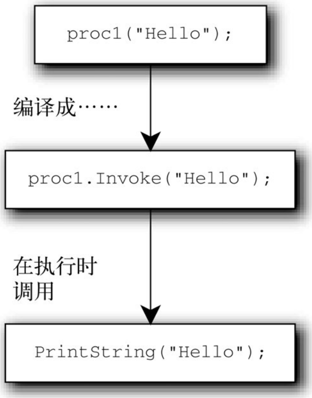


图2-1　处理使用C#简化语法的委托实例的调用

就是这么简单。所有原料都已齐备，接着将CLR预热到200℃，将所有东西都搅拌到一起，看看会发生什么。

5. 一个完整的例子和一些动机

通过一个完整的例子，可以看到操作中的全部内容——我们终于能真正运行一些东西了！由于有不少零碎的东西，所以这一次包含了完整的源代码，而不是使用“代码段”。在代码清单2-1中，没有什么令人兴奋的东西，所以不要期待惊喜——只是有了具体的代码可供讨论。

代码清单2-1　以各种简单的方式使用委托

```
using System;
delegate void StringProcessor(string input);        ❶ 声明委托类型
class Person
{
   string name;
   public Person(string name) { this.name = name; }

   public void Say(string message)
   {
      Console.WriteLine("{0} says: {1}", name, message);
   }                                                  ❷ 声明兼容的实例方法
}
class Background
{                                                     ❸ 声明兼容的静态方法
   public static void Note(string note)
   {
      Console.WriteLine("({0})", note);
   }
}
class SimpleDelegateUse
{
   static void Main()
   {
```

```
        Person jon = new Person("Jon");
        Person tom = new Person("Tom");
        StringProcessor jonsVoice, tomsVoice, background;    ❹ 创建3个委托实例
        jonsVoice = new StringProcessor(jon.Say);
        tomsVoice = new StringProcessor(tom.Say);
        background = new StringProcessor(Background.Note);
        jonsVoice("Hello, son.");                          ❺ 调用委托实例
        tomsVoice.Invoke("Hello, Daddy!");
        background("An airplane flies past.");
    }
}
```

首先声明委托类型❶，接着创建两个方法（❷和❸），它们都与委托类型兼容。一个是实例方法（Person.Say），另一个是静态方法（Background.Note），这样就可以看到在创建委托实例时❹，它们在使用方式上的区别。代码清单2-1创建了Person类的两个实例，便于观察委托目标所造成的差异。

jonsVoice被调用时❺，它会调用name为Jon的那个Person对象的Say方法。同样，tomsVoice被调用时，使用的是name为Tom的对象。这里只是出于兴趣，才展示了调用委托实例的两种方式——显式调用Invoke和使用C#的简化形式。一般情况下只需使用简化形式。

代码清单2-1的输出在预料之中：

```
Jon says: Hello, son.
Tom says: Hello, Daddy!
(An airplane flies past.)
```

坦白讲，如果仅仅是为了显示上述3行输出，代码清单2-1的代码未免太多了。即使想要使用Person类和Background类，也没有必要使用委托。那么，要点在哪里？为什么不直接调用方法？答案存在于我们最开始那个让律师执行遗嘱的例子中，不能仅仅由于你希望某事发生，就意味着你始终会在正确的时间和地点出现，并亲自使之发生。有时，你需要给出一些指令，将职责委托给别人。

应该强调的一点是，在软件世界中，没有对象"留遗嘱"这样的事情发生。经常都会发现这种情况：委托实例被调用时，最初创建委托实例的对象仍然是"活蹦乱跳"的。相反，委托相当于指定一些代码在特定的时间执行，那时，你也许已经无法（或者不想）更改要执行的代码。如果我希望在单击一个按钮后发生某事，但不想对按钮的代码进行修改，我只是希望按钮调用我的某个方法，那个方法能执行恰当的操作。委托的实质是间接完成某种操作，事实上，许多面向对象编程技术都在做同样的事情。我们看到，这增大了复杂性（看看为了输出这点儿内容，用了多少行代码），但同时也增加了灵活性。

现在已经对简单委托有了更多的理解，接着看看如何将委托合并到一起，以便成批地执行操作，而不是只执行一个。

2.1.2　合并和删除委托

到目前为止，我们见过的所有委托实例都只有一个操作。但真实的情况要稍微复杂一些：委托实例实际有一个操作列表与之关联。这称为委托实例的调用列表（invocation list）。

System.Delegate类型的静态方法Combine和Remove负责创建新的委托实例。其中，Combine负责将两个委托实例的调用列表连接到一起，而Remove负责从一个委托实例中删除另一个实例的调用列表。

说明 **委托是不易变的**　创建了委托实例后，有关它的一切就不能改变。这样一来，就可以安全地传递委托实例的引用，并把它们与其他委托实例合并，同时不必担心一致性、线程安全性或者是否有其他人试图更改它。在这一点上，委托实例和string是一样的。string的实例也是不易变的。之所以提到string，是因为Delegate.Combine和String.Concat很像——都是合并现有的实例来形成一个新实例，同时根本不更改原始对象。对于委托实例，原始调用列表被连接到一起。注意，如果试图将null和委托实例合并到一起，null将被视为带有空调用列表的一个委托。

很少在C#代码中看到对Delegate.Combine的显式调用，一般都是使用+和+=操作符。图2-2展示了转换过程，其中x和y都是相同（或兼容）委托类型的变量。所有转换都由C#编译器完成。

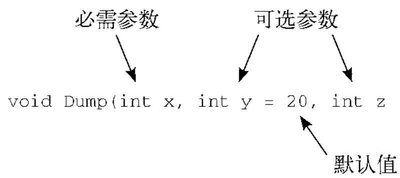


图2-2　用C#简化语法来合并委托实例时，C#编译器所执行的转换过程

可以看出，这是一个相当简单的转换过程，但它使代码变得整洁多了。除了能合并委托实例，还可以使用Delegate.Remove方法从一个实例中删除另一个实例的调用列表。C#使用-和-=运算符简写形式的方法非常简单，一看便知。Delegate.Remove(source, value)将创建一个新的委托实例，其调用列表来自source，value中的列表则被删除。如果结果有一个空的调用列表，就返回null。

调用委托实例时，它的所有操作都顺序执行。如果委托的签名具有一个非void的返回类型，则Invoke的返回值是最后一个操作的返回值。很少有非void的委托实例在它的调用列表中指定多个操作，因为这意味着其他所有操作的返回值永远都看不见。除非每次调用代码使用Delegate.GetInvocationList获取操作列表时，都显式调用某个委托。

如果调用列表中的任何操作抛出一个异常，都会阻止执行后续的操作。例如，假定调用一个委托实例，它的操作列表是[a, b, c]，但操作b抛出了一个异常，这个异常会立即“传播”，操作c不会执行。

进行事件处理时，委托实例的合并与删除会特别有用。既然我们已经理解了合并与删除涉及的操作，就很容易理解事件。

2.1.3　对事件的简单讨论

你可能对事件有了一个直观的概念，尤其是写过一些UI。它的基本思想是让代码在发生某事时作出响应，如在正确单击一个按钮后保存一个文件。在这个例子中，事件是“单击按钮”，操作是“保存文件”。然而，仅仅理解了一个概念的缘起，并不等同于理解了C#具体如何用语言来定义事件。

开发者经常将事件和委托实例，或者将事件和委托类型的字段混为一谈。但它们之间的差异十分大：事件不是委托类型的字段。之所以产生混淆，原因和以前相同，因为C#提供了一种简写方式，允许使用字段风格的事件（field-like event）。稍后就会讲到这种简写方式，但在此之前，先从C#编译器的角度看看事件到底由什么组成。

我认为将事件看做类似于属性（property）的东西是很有好处的。首先，两者都声明为具有一种特定的类型。对于事件来说，必须是一个委托类型。

使用属性时，感觉就像是直接对它的字段进行取值和赋值，但你实际是在调用方法，也就是取值方法和赋值方法[①]。实现属性时，可以在那些方法中做任何事情。但凑巧的是，大多数属性都只是实现了简单的字段，有时会在赋值方法中添加一些校验机制，有时则会添加一些线程安全性。

同样，在订阅或取消订阅一个事件时，看起来就像是在通过`+=`和`-=`运算符使用委托类型的字段。但和属性的情况一样，这个过程实际是在调用方法（`add`和`remove`方法）[②]。对于一个纯粹的事件，你所能做的事情就是订阅（添加一个事件处理程序）或者取消订阅（删除一个事件处理程序）。最终是由事件方法来做真正有用的事情，如找到你试图添加和删除的事件处理程序，并使它们在类中的其他地方可用。

“事件”存在的首要理由和“属性”差不多——它们添加了一个封装层，实现发布/订阅模式（publish/subscribe pattern），参见我的文章“Delegates and Events,” 网址是http://mng.bz/HPx6。通常，我们不希望其他代码能直接设置字段值；最起码也要先由所有者（owner）对新值进行验证。同样，我们通常不希望类外部的代码随意更改（或调用）一个事件的处理程序。当然，类能通过添加方法的方式来提供额外的访问。例如，可以重置事件的处理程序列表，或者引发事件（也就是调用它的事件处理程序）。例如，`BackgroundWorker.OnProgressChanged`只是调用了`ProgressChanged`事件的处理程序。然而，如果只对外揭示事件本身，类外部的代码就只能添加和删除事件处理程序。

字段风格的事件使所有这些的实现变得更易阅读，只需一个声明就可以了。编译器会将声明转换成一个具有默认`add`/`remove`实现的事件和一个私有委托类型的字段。类内的代码能看见字

① 取值方法也可以说成是get accessor或者获取方法；而赋值方法也可以说成是set accessor或者设置方法。——译者注

② 在编译好的代码中，它们的名称不是add和remove；否则的话，每个类型就只能使用一个事件了。编译器会使用不能在别处使用的名称来创建两个方法。另外，还会创建一些特殊的元数据（metadata），使其他类型知道现在有一个指定名称的事件，而且知道它们所调用的`add`/`remove`方法。

段；类外的代码只能看见事件。这样一来，表面上似乎能调用一个事件，但为了调用事件处理程序，实际做的事情是调用存储在字段中的委托实例。

事件的细节超出了本章的范围——事件本身在更高版本的C#[①]里没有多大变化，但我希望现在就强调一下委托实例和事件的差异，以免日后混淆。

2.1.4　委托总结

下面对委托进行总结：

- 委托封装了包含特殊返回类型和一组参数的行为，类似包含单一方法的接口；
- 委托类型声明中所描述的类型签名决定了哪个方法可用于创建委托实例，同时决定了调用的签名；
- 为了创建委托实例，需要一个方法以及（对于实例方法来说）调用方法的目标；
- 委托实例是不易变的；
- 每个委托实例都包含一个调用列表——一个操作列表；
- 委托实例可以合并到一起，也可以从一个委托实例中删除另一个；
- 事件不是委托实例——只是成对的`add`/`remove`方法（类似于属性的取值方法/赋值方法）。

委托是C#和.NET的一个非常具体的主题，是大背景下的一个小细节。在本章剩余的部分，将讨论一些更宽泛的主题。首先要讨论的是，当说到C#是一种*静态类型*的语言时，具体是什么意思，以及这种说法有何暗示。

2.2　类型系统的特征

几乎每种编程语言都有某种形式的类型系统。随着时间的推移，这种类型系统被分为强/弱、安全/不安全、静态/动态以及其他一些让人更不好懂的说法。理解各种类型系统与哪种语言一起使用，显然很重要。另外，期望了解哪种语言属于哪种类型系统，以获得大量信息，从而在这方面有所帮助，这是很合理的。然而，由于不同的人经常用不同的术语来指代差别不是太大的两种东西，所以很容易产生沟通障碍。我会尝试解释每个术语到底指的什么，以尽可能地避免这样的混淆。

要注意的一个重点在于，本节只适用于*安全代码*，也就是没有被显式放到一个不安全（unsafe）上下文中的所有C#代码。从名字就可以判断出，不安全上下文中的代码能做安全代码不能做的各种事情，尽管类型系统在其他许多方面仍然安全，但这种操作可能违反正常的类型安全性（type safety）的一些准则。大多数开发者平时都用不着写不安全代码，另外，如果只考虑安全代码，那么类型系统的各种特征会变得更容易描述和理解。

本节描述了在C# 1中什么是限制，以及哪些限制没有实行。同时，我们定义了一些术语来描述对应的行为。然后，我们介绍了用C# 1不能做的几件事情——首先解释了哪些事情*没有*办法告诉编译器，然后解释了我们希望哪些事情*不用*告诉编译器。

① C# 4对字段风格的事件进行了很小的改进。详细内容参见4.2节。

首先从C# 1能做的事情讲起，以及要用哪些术语来描述这种行为。

2.2.1　C#在类型系统世界中的位置

为了理解这个主题，最容易的办法是先进行说明，再解释它的含义以及可供选择的内容：

C# 1的类型系统是静态的、显式的和安全的。

你可能很期望强这个形容词出现在列表中。我有点儿想包括它。但是，虽然大多数人都很容易就一种语言是否具有上面列出的特征取得共识，但在判断语言是不是一种强类型的语言时，却可能引发激烈争论。这是由于“强类型”存在多种不同的定义。在某些“强类型”定义中，要求禁止任何形式的转换（不管是显式还是隐式转换），这明显会使C#失去资格。但在另一些定义中，却相当接近（甚至等同于）静态类型，这会使C#获得资格。我读过的大多数文章和书籍都将C#描述成强类型的语言，但最终的意思实际都是指它是一种静态类型的语言。

现在，让我们依次阐述上述结论中的每一个术语。

1. 静态类型和动态类型

C#是静态类型的：每个变量都有一个特定的类型，而且该类型在编译时是已知的[①]。只有该类型已知的操作才是允许的，这一点由编译器强制生效。来看下面这个强制生效的例子：

```
object o = "hello";
Console.WriteLine(o.Length);
```

从开发者的角度看上述代码，我们知道o的值是一个字符串。同时，string类型有一个Length属性。但是，编译器只把o看做object类型。如果想访问Length属性，必须让编译器知道o的值实际是一个字符串：

```
object o = "hello";
Console.WriteLine(((string)o).Length);
```

接下来编译器会查找System.String的Length属性。以此来验证调用是否正确，生成适当的IL，并计算出整个表达式的类型。表达式的编译时类型仍然为静态类型，因此我们可以说，“o的静态类型为System.Object”。

说明　**为什么称为静态类型？**　静态这个词用来描述表达式的编译时类型，因为它们使用不变的（unchanging）数据来分析哪些操作可用。假设某个变量声明为Stream类型，那么不管变量的值为MemoryStream还是FileStream，甚至不是流类型（包括空引用），它的类型都不会发生改变。然而静态类型系统中也可以有一些动态行为，如虚方法调用所执行的实际实现依赖于所调用的对象。尽管这种“不变信息”的理念也是static修饰符的背后动机，但简便起见还是应该认为静态成员属于类型本身，而不属于某个类型的特定实例。对于多数实际应用，可以认为该单词的这两种用法毫无关系。

---

① 也适合大多数（但并非所有）表达式。有的表达式没有类型，比如void方法调用，但这不影响C# 1的静态类型特征。本节会一直使用“变量”（variable）一词，以避免无谓的冲突。

与静态类型对应的是动态类型，后者可能具有多种形式。动态类型的实质是变量中含有值，但那些值并不限于特定的类型，所以编译器不能执行相同形式的检查。相反，执行环境试图采取一种合适的方式来理解引用值的给定表达式。例如，假定C# 1是动态类型的，就可以做下面的事情：

```
o = "hello";
Console.WriteLine(o.Length);
o = new string[] {"hi", "there"};
Console.WriteLine(o.Length);
```

通过在执行时动态检查类型，最终会调用两个完全无关的Length属性——String.Length和Array.Length。和许多定义类型系统的区域一样，动态类型具有不同的级别。有的语言允许在你希望的任何地方指定类型——除了在赋值时指定，指定的类型可能仍被当作动态类型处理——但在其他地方仍然使用没有指定类型的变量。

尽管我多次声明这是C# 1，但直到C# 3时它还是一门完全静态的语言。稍后我们将看到，C# 4引入了动态类型，然而大多数C# 4应用程序中的大部分代码仍然是静态类型的。

2. 显式类型和隐式类型

显式类型和隐式类型的区别只有在静态类型的语言中才有意义。对于显式类型来说，每个变量的类型都必须在声明中显式指明。隐式类型则允许编译器根据变量的用途来推断变量的类型。例如，语言可以推断出变量的类型是用于初始赋值的那个表达式的类型。

以一个假设的语言为例，它用关键字var告诉编译器进行类型推断[①]。表2-1展示了这种语言的代码在C# 1中要如何写。左边一列的代码在C# 1中是不允许的，但是，右边一列是等价的有效代码。

表2-1　展示隐式类型和隐式类型区别的一个例子

无效的C# 1——隐式类型	有效的C# 1——显式类型

`var s = "hello";`
`var x = s.Length;`
`var twiceX = x * 2;`
	`string s = "hello";`
	`int x = s.Length;`
	`int twiceX = x * 2;`

为什么这个例子只是关于静态类型的情况？答案一目了然：无论隐式类型还是显式类型，变量的类型在编译时都是已知的——即使在代码中没有显式地声明。在动态类型的情况下，变量根本没有一个类型可供声明或推断。

3. 类型安全与类型不安全

描述类型安全系统的最简单方法就是描述它的对立面。有的语言（我认为尤其是C和C++）允许做一些非常“不正当”的事情。但在合适的时候，其功能可能会很强大。但是，世界上没有免费的午餐。所谓“合适的时候”实际很少能够遇到。如使用不当，反而极有可能“搬起石头砸自己的脚”。滥用类型系统就属于这种情况。

使用一些非正当的方法，可以使语言将一种类型的值当作另一种完全不同的类型的值，同时不必进行任何转换。我并不只是说像前面动态类型的例子那样调用一个方法，而该方法与我们想

① 好吧，其实也不算假设，请参见8.2节，了解C# 3的隐式类型本地变量功能。

调用的方法名称相同。我的意思是说,有的代码会以错误的方式检查值中的原始字节并解释它们。代码清单2-2展示了一个简单的C的例子。

代码清单2-2　使用C代码来演示不安全类型的系统

```
#include <stdio.h>
int main(int argc, char**argv)
{
   char *first_arg = argv[1];
   int *first_arg_as_int = (int *)first_arg;
   printf ("%d", *first_arg_as_int);
}
```

如果你编译并运行代码清单2-2，并传递一个简单的参数值`"hello"`，最后显示的值是`1819043176`——至少，如果在一个小端序（little-endian）的架构中，编译器将`int`当作32位值，将`char`当作8位值，而且文本用ASCII或UTF-8来表示。代码将`char`指针视为一个`int`指针，所以取其返回文本的前4字节，并把它们视为一个数字。

事实上，这个小例子和另一种可能的滥用比起来还算不上什么。在完全无关的结构（struct）之间进行强制类型转换，很容易造成严重的后果。这不仅在现实生活中经常发生，而且C类型系统的一些元素也要求你必须告诉编译器做什么，使它只能无条件地相信你——即使是在执行时。

幸好，所有这些在C#中都不会发生。是的，可以进行大量转换，但不能自欺欺人地说一种类型的数据是全然不同的另一种类型的数据。可以试着添加一个强制类型转换，为编译器提供这种额外的（和不正确的）信息。但是，假如编译器发现这种转换实际是不可能的，就会触发一个编译时错误。另外，如果理论上允许，但在执行时发现不正确，CLR也会抛出一个异常。

既然我们知道了C# 1如何适应类型系统中的位置，接着应该说一下它选择时的几个缺点。这并不是说它对于类型系统的选择是错误的，只是说在某些方面确实存在着局限。通常，语言的设计者在权衡不同的设计方式时，必须权衡每一种方式存在的限制或者其他不利因素。首先来看看这样一种情况：你希望告诉编译器更多的信息，但却没办法做到。

2.2.2　C# 1 的类型系统何时不够用

在两种常见的情况下，你可能想向方法的调用者揭示更多的信息，或者想强迫调用者对它们在参数值中提供的内容进行限制。第一种情况涉及集合，第二种情况涉及继承和覆盖方法或实现接口[①]。我们将依次进行讨论。

1. 集合，强和弱

通常，应该避免使用“强”和“弱”来描述C#类型系统。但在讨论集合时，我打算使用它们。在“集合”的上下文中，几乎所有地方都要使用这两个词，而且几乎不可能产生歧义。一般来说，.NET 1.1内建了以下3种集合类型：

- 数组——强类型——内建到语言和运行时中；

① 也就是从基类或接口继承时，覆盖基类定义的virtual方法，或者实现从接口继承的方法。——译者注

- System.Collections命名空间中的弱类型集合；
- System.Collections.Specialized命名空间中的强类型集合。

数组是强类型的[①]。所以在编译时，不可能将string[]的一个元素设置成一个FileStream。然而，引用类型的数组也支持协变（covariance），只要元素的类型之间允许这样的转换，就能隐式将一种数组类型转换成另一种类型。执行时会进行检查，以确保类型有误的引用不会被存储下来，如代码清单2-3所示。

代码清单2-3 数组协变以及执行时类型检查的演示

```
string[] strings = new string[5];
object[] objects = strings;        ❶ 应用协变式转换
objects[0] = new Button();         ❷ 试图存储一个按钮引用
```

运行代码清单2-3，会抛出一个ArrayTypeMismatchException异常❷。这是由于从string[]转换成object[]❶会返回原始引用，无论strings还是objects都引用同一个数组。数组本身“知道”它是一个字符串数组，所以会拒绝存储对非字符串的引用。数组协变有时会派上用场，但代价是一些类型安全性在执行时才能实现，而不能在编译时实现。

让我们把它与弱类型的集合（如ArrayList和Hashtable）的情况进行对比。这些集合的API定义键和值的类型是object。例如，写一个ArrayList方法时，没有办法保证在编译时调用者会传入一个字符串列表。可以把这个要求写入文档。只要将列表的每个元素都强制转换为string，运行时的类型安全性就会帮你强制使这个限制生效。但是，这样不会获得编译时的类型安全性。同样，如果返回一个ArrayList，可以在文档中指出它只包含字符串。但是，调用者必须相信你说的是实话，而且在访问列表元素时必须插入强制类型转换。

最后看看强类型的集合，如StringCollection。这些集合提供了一个强类型的API。所以，如果接受一个StringCollection作为参数或返回值，可以肯定它只包含string。另外，在取回集合的元素时，不需要进行强制类型转换。这听起来似乎很理想，但有两个问题。首先，它实现了IList，所以仍然可以为它添加非字符串（的对象），虽然运行时会失败。其次，它只能处理字符串。还有一些专门的集合，但它们包括的范围不是很大。例如，CollectionBase类型，可以用它构建你自己的强类型集合，但那意味着要为每种元素类型都创建一个新集合，所以同样不理想。

既然我们已经知道了集合的问题，接着看看覆盖方法和实现接口时发生的问题。它们和协变的概念有关，前面讲数组时已经讲过这个概念。

2. 缺乏协变的返回类型

ICloneable是框架中最简单的接口之一。它只有一个Clone方法，该方法返回调用方法的那个对象[②]的一个副本。暂不讨论这应该是一个深拷贝还是一个浅拷贝，先看看Clone方法的签名：

① 至少语言允许它们是强类型的。然而，使用Array类型可以对数组进行弱类型的访问。

② 或者说“在上面调用该方法的那个对象”；在英语文档中，将“对象.方法”这样的情况称为在对象上调用方法。——译者注

```
object Clone()
```

这是一个非常简单的签名。就像刚才所说，该方法应返回调用方法的那个对象的一个副本。这意味着它需要返回同类型的一个对象，或至少兼容类型的一个对象（具体含义要取决于类型）。

用一个覆盖方法的签名更准确地描述该方法实际的返回值，应该讲得通。例如，在Person类中，像下面这样实现ICloneable接口是不错的选择：

```
public Person Clone()
```

这应该破坏不了任何东西，代码期待的旧的对象仍然能够正常工作。这个特性称为返回类型的协变性。但遗憾的是，接口实现和方法覆盖不支持这一特性。对于接口来说，正常的解决方法是使用*显式接口实现*（explicit interface implementation）来获得预期的效果。

```
public Person Clone()
{
  [Implementation goes here]
}
object ICloneable.Clone()            <—— 显式实现接口
{
   return Clone();              <—— 调用非接口方法
}
```

这样一来，任何代码为一个表达式调用Clone()时，如果编译器知道这个表达式的类型是Person，就会调用上面的方法；如果表达式的类型只是ICloneable，就会调用下面的方法。这虽然可行，但真的太别扭了。参数也存在类似的问题。假定一个接口方法或一个虚方法，其签名是void Process(string x)，那么在实现或者覆盖这个方法时，使用一个放宽了限制的签名应该是合乎逻辑的，如void Process(object x)。这称为*参数类型的逆变性*（parameter type contravariance）。但是，和返回类型的协变性一样，参数类型的逆变性也是不支持的。那么应该如何解决？对于接口，解决方案是一样的，同样都是进行显式接口实现。对于虚方法，解决方案则是进行普通的方法重载。虽然不是什么大问题，却着实烦人。

当然，C# 1的开发者被这些问题折磨了很长时间，Java开发者的情况类似，而且他们被折磨了更长时间。虽然编译时的类型安全性总的来说是非常出色的一个特性，但许多bug实际上是由于在集合中放置了错误类型的元素造成的。我已经记不清见过多少这样的bug了。在没有协变性和逆变性时，虽然日子一样可以过，但我们写程序时，还有一个目标是“优雅”。换言之，代码应清楚地表述我们的意思，最好*不需要*附加注释就可以让人明白。尽管可能并不会实际产生bug，但在文档契约中强制规定一些内容（如集合只能包含字符串），也是昂贵且脆弱的（对于易变集合来说）。我们真的希望类型系统本身能够遵循这样的契约。

以后会看到，C# 2在这方面也不是完美的，但它确实有了相当大的改进。C# 4的改进更大，但还是不包括返回类型协变和参数逆变[①]。

① C# 4引入了受限的泛型协变和逆变，但这与我们所说的不是一回事。

2.2.3 类型系统特征总结

本节描述了不同类型系统的一些差异，并具体描述了C# 1的特征：

- C# 1是静态类型的——编译器知道你能使用哪些成员；
- C# 1是显式的——必须告诉编译器变量具有什么类型；
- C# 1是安全的——除非存在真实的转换关系，否则不能将一种类型当做另一种类型；
- 静态类型仍然不允许一个集合成为强类型的“字符串列表”或者“整数列表”，除非针对不同的元素使用大量的重复代码；
- 方法覆盖和接口实现不允许协变性/逆变性。

下一节暂停讨论C#类型系统的高级特征。相反，让我们讨论一下最基本的概念：结构和类的差异。

2.3 值类型和引用类型

再怎么强调本节主题的重要性都不为过！在.NET中做的一切其实都是在和值类型或者引用类型打交道。但极有可能一些人即使使用C#开发了很长一段时间，对这些差异也只是有一个模糊的概念。更糟的是，可能还存在着一些误解，导致局面变得更复杂。遗憾的是，稍不留神，就很容易作出一个简短但不正确的陈述，它非常接近真相，所以看起来似乎是真的；但又不够准确，从而对人产生了误导。不过，要用简洁而准确的一段描述来解释两者的差异也不是一件容易的事情。

本节不打算全面分析类型如何处理，如何在不同的应用程序域（application domain）之间封送（marshaling），如何与本地代码互操作，等等。相反，我们只是简要讨论了为了深入更高版本C#的世界，C# 1的哪些主题的基本元素是必须理解的。

先来看看在现实世界和在.NET中，值类型和引用类型的基本差异是如何自然体现的。

2.3.1 现实世界中的值和引用

假定你在读一份非常棒的东西，希望一个朋友也去读它。为了避免被人投诉支持盗版，进一步假定它是公共领域中的一份文档。那么，需要为朋友提供什么才能让他读到文档呢？这完全取决于阅读的内容。

先假设你正在读的是一份真正的报纸。为了给朋友一份，需要影印报纸的全部内容并交给他。届时，他将获得属于他自己的一份完整的报纸。在这种情况下，我们处理的是值类型的行为。所有信息都在你的手上，不需要从任何其他地方获得。制作了副本之后，你的这份信息和朋友的那份是各自独立的。可以在自己的报纸上添加一些注解，他的报纸根本不会改变。

再假设你正在读的是一个网页。与前一次相比，这一次，唯一需要给朋友的就是网页的URL。这是引用类型的行为，URL代替引用。为了真正读到文档，必须在浏览器中输入URL，并要求它加载网页来导航引用。另一方面，假如网页由于某种原因发生了变化（如一个维基页面，你在上面添加了自己的注释），你和你的朋友下次载入页面时，都会看到那个改变。

在C#和.NET中，值类型和引用类型的差异与现实世界中的差别类似。.NET中的大多数类型都是引用类型，你以后创建的引用类型极有可能比值类型多很多。除了以下总结的特殊情况，类（使用class来声明）是引用类型，而结构（使用struct来声明）是值类型。特殊情况包括如下方面：

- 数组类型是引用类型，即使元素类型是值类型（所以即便int是值类型，int[]仍是引用类型）；
- 枚举（使用enum来声明）是值类型；
- 委托类型（使用delegate来声明）是引用类型；
- 接口类型（使用interface来声明）是引用类型，但可由值类型实现。

理解了引用类型和值类型的基本概念之后，接着要探讨几个最重要的细节。

2.3.2　值类型和引用类型基础知识

学习值类型和引用类型时，要掌握的重要概念是一个特殊表达式的值是什么。为使问题更加具体，我使用了表达式最常见的例子——变量。但是，同样的道理也适用于属性、方法调用、索引器（indexer）和其他表达式。

2.2.1节说过，大多数表达式都有与其相关的静态类型。对于值类型的表达式，它的值就是表达式的值，这很容易理解。例如，表达式“2+3”的值就是5。然而，对于*引用*类型的表达式，它的值是一个引用，而*不是*该引用所指代的对象。所以，表达式String.Empty的值不是一个空字符串——而是对空字符串的一个引用。在平常的讨论中，甚至在一些专业的文档中，经常混淆这一区别。例如，你可能这样描述：“String.Concat的作用是返回一个字符串，该字符串将所有参数都连接到了一起。”如果在这里使用非常精确的术语，既花时间又分散注意力。只要每个人都理解返回的只是一个*引用*，就没有问题。

为了进一步演示这个问题，来看看存储了两个整数x和y的一个Point类型，它的一个构造函数能获得两个值。现在，这个类型可以实现为结构或类。图2-3展示了执行下面两行代码的结果：

```
Point p1 = new Point(10, 20);
Point p2 = p1;
```

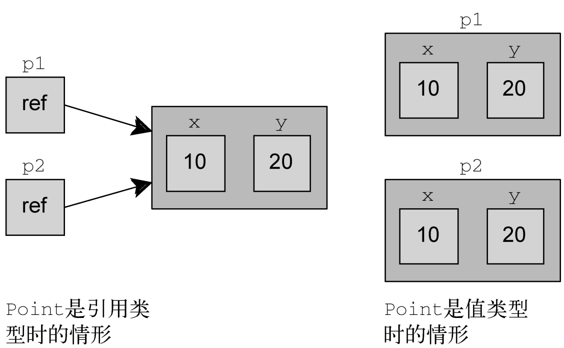


图2-3　比较值类型和引用类型的行为，尤其是涉及赋值操作时

图2-3左边的部分指出当Point是一个类（引用类型）时所涉及的值，右边的部分展示了当Point是一个结构（值类型）时的情形。在两种情况下，p1和p2在赋值后都有相同的“值”。然而，在Point是引用类型的情况下，那个“值”是引用：p1和p2都引用同一个对象。在Point是值类型的情况下，p1的值是一个“点”的完整的数据，也就是这个“点”的x和y值。将p1的值赋给p2，会复制p1的所有数据。

变量的值在它声明时的位置存储。局部变量的值总是存储在栈（stack）中①，实例变量的值总是存储在实例本身存储的地方。引用类型实例（对象）总是存储在堆（heap）中，静态变量也是。

两种类型的另一个差异在于，值类型不可以派生出其他类型。这将导致的一个结果就是，值不需要额外的信息来描述值实际是什么类型。把它同引用类型比较，对于引用类型来说，每个对象的开头都包含一个数据块，它标识了对象的实际类型，同时还提供了其他一些信息。永远都不能改变对象的类型——执行简单的强制类型转换时，运行时会获取一个引用，检查它引用的对象是不是目标类型的一个有效对象。如果有效，就返回原始引用；否则抛出异常。引用本身并不知道对象的类型——所以同一个引用“值”可用于（引用）不同类型的多个变量。例如下面的代码：

```
Stream stream = new MemoryStream();
MemoryStream memoryStream = (MemoryStream) stream;
```

第1行创建一个新的MemoryStream对象，并将stream变量的值设为对那个新对象的引用。第2行检查stream的值引用的是不是一个MemoryStream（或派生类型）对象，并将MemoryStream的值设为相同的值。

理解了这些基本知识点后，平时看到对值类型及引用类型的错误描述时，就可应用这些知识来解决问题。

2.3.3　走出误区

经常都会听到各式各样的错误说法。我可以确定，这些错误的信息会长时间地流传下去。许多人会不自觉地接受它们，根本意识不到自己的认识其实已经出现了偏差。本节将处理一些典型错误，解释真实的情况到底是什么。

误区1：“结构是轻量级的类”

这个误解存在着多种形式。有人认为值类型不能或不应有方法或其他有意义的行为——它们应作为简单的数据转移类型来使用，只应该有public字段或简单的属性。对于这种说法，一个非常典型的反例就是DateTime类型：它作为值类型来提供是很有道理的，因为它非常适合作为和数字或字符相似的一个基本单位来使用。另外，它也理应被赋予对它的值执行计算的能力。换个角度来看这个问题，是数据转移类型一般都是引用类型。总之，具体应该如何决定，应取决于需要的是值类型的语义，还是引用类型的语义，而不是取决于这个类型简单与否。

还有一些人认为值类型之所以显得比引用类型“轻”，是因为性能。事实是在某些情况下，

① 这个结论只有在C# 1中完全成立。以后会讲到，在更高版本C#中，在特定情况下，局部变量最终可能存储在堆中。

值类型很“能干”——它们不需要垃圾回收，(除非被装箱)不会因类型标识而产生开销，也不需要解引用。但在其他方面，引用类型显得更“能干”——在传递参数、赋值、将值返回和执行类似的操作时，只需复制4或8字节(要看运行的是32位还是64位CLR)，而不是复制全部数据。假定ArrayList是一个所谓“纯的”值类型，那么将一个ArrayList表达式传给一个方法时，就得复制它的所有数据！几乎在所有情况下，性能问题都不是根据这种判断来决定的。瓶颈从来都不是想当然的，在你根据性能进行设计之前，需要衡量不同的选择。

值得注意的是，将这两者相结合也不能解决问题：类型(不管是类还是结构)拥有多少方法并不重要，每个实例所占用的内存不会受到影响。(代码本身会消耗内存，但这只会发生一次，而不是每个实例都发生。)

误区2：“引用类型保存在堆上，值类型保存在栈上”

这个误区主要应归咎于转述这句话的人根本没有动脑筋。第一部分是正确的——引用类型的实例总是在堆上创建的。但第二部分就有问题了。前面讲过，变量的值是在它声明的位置存储的。所以，假定一个类中有一个int类型的实例变量，那么在这个类的任何对象中，该变量的值总是和对象中的其他数据在一起，也就是在堆上。只有局部变量(方法内部声明的变量)和方法参数在栈上。对于C# 2及更高版本，很多局部变量并不完全存放在栈中，第5章中的匿名方法会讲到这一点。

说明　这些概念符合现实吗？　一个较有争议的说法是：写托管代码时，应该让运行时去操心内存的最佳使用方式。事实上，在语言规范中，并没有对什么东西应该存储在什么地方做出硬性规定。在未来的某个版本的运行时中，也许会允许在栈上创建一些对象(前提是运行时知道这样可行)。又或者，C#编译器能生成几乎完全用不到栈的代码。

下一个误区是由于对术语的理解出现了偏差。

误区3：“对象在C#中默认是通过引用传递的”

这或许是传播得最广的一个误区了。同样，说这句话的人一般(但并不总是)知道C#实际的行为是什么，但不知道“引用传递”(pass by reference)的真正意思是什么。可惜，那些真正知道引用传递是什么意思的人，在听到这句话时就会被完全搞糊涂。

“引用传递”的正式定义相当复杂，要涉及左值(l-values)和类似的计算机科学术语。但最重要的一点是，假如以引用传递的方式来传送一个变量，那么调用的方法可以通过更改其参数值，来改变调用者的变量值。现在请记住，引用类型变量的值是*引用*，而不是对象本身。不需要按引用来传递参数本身，就可以更改该参数引用的那个对象的*内容*。例如，下面的方法更改了相关对象StringBuilder的内容，但调用者的表达式引用的仍然是之前的那个对象：

```
void AppendHello(StringBuilder builder)
{
   builder.Append("hello");
}
```

调用这个方法时，参数值(对StringBuilder的一个引用)是以值传递(pass by value)的方式传递的。如果想在方法内部更改builder变量的值——如执行builder = null;语句，调

用者看不见这个改变[①]，刚好跟错误认识相反。

有趣的是，在这种错误说法中，不仅“引用传递”的说法有误，而且“对象传递”的说法也存在问题。无论是引用传递还是值传递，永远不会传递对象本身。涉及一个引用类型时，要么以“引用传递”的方式传递变量，要么以“传值”的方式传递参数值（引用）。最起码，这回答了“当null作为一个传值参数的值来使用时会发生什么”的问题。假如传递的是对象，这时就会出问题，因为没有一个对象可供传递！相反，null引用会采用和其他引用一样的“值传递”的方式传递。

如果这种简要的解释还让你感到困惑，可以浏览我的文章“Parameter passing in C#”（http://mng.bz/otVt），那里有更详细的介绍。

除此之外，还有另一些误解。许多人对于装箱和拆箱的理解也存在一定的误区，下一节将对此加以澄清。

2.3.4 装箱和拆箱

有时，我们就是不想用值类型的值，就是想用一个引用。之所以会发生这种情况，有多种原因。幸好，C#和.NET提供了一个名为装箱（boxing）的机制，它允许根据值类型创建一个对象，然后使用对这个新对象的一个引用。在接触实际的例子之前，先来回顾两个重要的事实：

- 对于引用类型的变量，它的值永远是一个引用；
- 对于值类型的变量，它的值永远是该值类型的一个值。

基于这两个事实，下面3行代码第一眼看上去似乎没有太多道理：

```
int i = 5;
object o = i;
int j = (int) o;
```

这里有两个变量：i是值类型的变量，o是引用类型的变量。将i的值赋给o有道理吗？o的值必须是一个引用，而数字5不是引用，它是一个整数值。实际发生的事情就是装箱：运行时将在堆上创建一个包含值（5）的对象（它是一个普通对象）。o的值是对该新对象的一个引用。该对象的值是原始值的一个副本，改变i的值不会改变箱内的值。

第3行执行相反的操作——拆箱。必须告诉编译器将object拆箱成什么类型。如果使用了错误的类型（比如o原先被装箱成unit或者long，或者根本就不是一个已装箱的值），就会抛出一个InvalidCastException异常。同样，拆箱也会复制箱内的值，在赋值之后，j和该对象之间不再有任何关系。

上面这一段话其实已经简单明了地解释了装箱和拆箱。剩下的唯一的问题就是要知道装箱和拆箱在什么时候发生。拆箱一般是很容易看出来，因为要在代码中明确地显示一个强制类型转换。装箱则可能悄悄进行。上例展示的是一个简单的版本（第2行代码）。但是，为一个类型的值调用ToString、Equals或GetHashCode方法时，如果该类型没有覆盖这些方法[②]，也会发生装箱。

① 这样更改的是builder值的一个副本，当然是调用者看不见的。——译者注

② 当你调用值类型变量的GetType()方法时总是伴随着装箱过程，因为它不能被重载。如果处理未装箱形式的变量，你应该已经知道了具体类型，因此使用typeof替代即可。

另外，将值作为接口表达式使用时——把它赋给一个接口类型的变量，或者把它作为接口类型的参数来传递——也会发生装箱。例如，`IComparable x = 5;`语句会对数字5进行装箱。

之所以要留意装箱和拆箱，是由于它们可能会降低性能。一次装箱或拆箱操作的开销是微不足道的，但假如执行千百次这样的操作，那么不仅会增大程序本身的操作开销，还会创建数量众多的对象，而这些对象会加重垃圾回收器的负担。同样，这种性能损失通常也不是大问题，但还是应该引起注意，因此如果你关心的话，可以对拆装箱带来的影响进行测试。

2.3.5　值类型和引用类型小结

本节讨论了值类型和引用类型的差异，还澄清了围绕它们存在的一些误区。下面是一些要点。

- 对于引用类型的表达式（如一个变量），它的值是一个引用，而非对象。
- 引用就像URL——是允许你访问真实信息的一小片数据。
- 对于值类型的表达式，它的值是实际的数据。
- 有时，值类型比引用类型更有效，有时恰好相反。
- 引用类型的对象总是在堆上，值类型的值既可能在栈上，也可能在堆上，具体取决于上下文。
- 引用类型作为方法参数使用时，参数默认是以“值传递”方式来传递的——但值本身是一个引用。
- 值类型的值会在需要引用类型的行为时被装箱；拆箱则是相反的过程。

既然我们已经讨论了你需要适应的C# 1的所有特性。接着简单看一下每个特性在更高版本的C#中得到了哪些增强。

2.4　C# 1 之外：构建于坚实基础之上的新特性

本章讨论的3个主题对于所有版本C#来说均十分重要。几乎所有新特性都至少同其中的一个主题有关。它们改变了对于语言使用方式的平衡。在结束本章之前，我们探讨一下新特性与旧特性之间的关联。这里不打算给出太多的细节（因为出版商不希望这一节的篇幅达到600页），但在讨论实际内容之前，了解一下这些领域的发展方向会很有帮助。下面将按照和前面一样的顺序讨论它们，首先从委托开始。

2.4.1　与委托有关的特性

各种委托在C# 2中都得到了增强，在C# 3中则获得了更特殊的待遇。大多数特性对于CLR来说都不是新内容，它们只是聪明的编译器变的一些“戏法”，使委托在语言中能够更顺利地工作。变化所影响的不仅仅是语法，还有C#惯用代码的外观和感受。随着时间的推移，C#获得了更加函数化的委托处理方式。

创建委托实例时，C# 1使用的语法是相当笨拙的。一方面，即使要做的事情非常简单，也必须要有一个专门做这件事情的方法，才能为这个方法创建一个委托实例。C# 2使用匿名方法对此

进行了修正。另外，还引入了一个简单的语法，用于仍然要用一个普通的方法为委托提供操作的情况。最后，还可以使用具有兼容签名的方法来创建委托实例——方法签名不需要和委托的声明完全一致。

代码清单2-4演示了这些改进。

代码清单2-4　演示C# 2在委托实例化上的改进

```
static void HandleDemoEvent(object sender, EventArgs e)
{
   Console.WriteLine ("Handled by HandleDemoEvent");
}
...
EventHandler handler;                                    ❶ 指定委托类型和方法
handler = new EventHandler(HandleDemoEvent);
handler(null, EventArgs.Empty);

handler = HandleDemoEvent;                               ❷ 隐式转换成委托实例
handler(null, EventArgs.Empty);

handler = delegate(object sender, EventArgs e)
{
   Console.WriteLine ("Handled anonymously");            ❸ 用一个匿名方法来指定操作
};
handler(null, EventArgs.Empty);

handler = delegate
{
   Console.WriteLine ("Handled anonymously again");      ❹ 使用匿名方法的简写形式
};
handler(null, EventArgs.Empty);

MouseEventHandler mouseHandler = HandleDemoEvent;        ❺ 使用委托逆变性
mouseHandler(null, new MouseEventArgs(MouseButtons.None,
                                      0, 0, 0, 0));
```

Main代码的第一部分❶是用C# 1写的代码，这里保留供对照。其他委托都使用了C# 2的新特性。方法组转换❷使事件订阅代码看起来更加友好——例如，saveButton.Click += SaveDocument;这样的代码看起来非常直观，没有难理解的内容。匿名方法的语法❸看起来要繁琐一些，但确实使要采取的操作变得更清晰。在操作的创建位置，就可以当场了解这个行动具体要干什么，不必转移注意力去看另一个方法才能了解具体发生的事情。使用匿名方法进行简写❹，但只有在不需要参数时才使用这种形式。匿名方法还有其他强大的功能，稍后将进行详述。

创建的最后一个委托实例❺是MouseEventHandler的实例，而不只是一个EventHandler实例。但是，HandleDemoEvent方法仍可使用，这是借助于*逆变性*来实现的，它指定了参数的兼容性。*协变性*指定了返回类型的兼容性。逆变性和协变性将在第5章详细讨论。事件处理程序也许是逆变性和协变性最大的受益者。突然间，在微软的指南[①]中，要求事件中使用的所有委托类型都遵循相同的约定，这是很有道理的。在C# 1中，两个不同的事件处理程序看起来是否"很相

① 请在MSDN中查找"Event Usage Guidelines"。——译者注

似”是没有关系的——方法必须具有完全匹配的签名，方能创建委托实例。在C# 2中，则可用同一个方法处理许多不同的事件，尤其是假如方法所做的事情在很大程度上独立于事件本身，如日志记录。

C# 3提供了一个特殊的语法来实例化委托类型，这就是使用Lambda表达式。为了对此进行演示，我们将使用一个新的委托类型。在.NET 2.0中，随着CLR在泛型上的增强，我们也可以使用泛型委托类型。在泛型集合的大量API调用中，都使用了这种泛型委托类型。然而，.NET 3.5更进一步，引入了一组名为Func的泛型委托类型，它能获取多个指定类型的参数，并返回另一个指定类型的值。代码清单2-5展示了使用Func委托类型以及Lambda表达式的例子。

代码清单2-5　Lambda表达式像是改进后的匿名方法

```
Func<int,int,string> func = (x, y) => (x * y).ToString();
Console.WriteLine(func(5, 20));
```

Func<int,int,string>是一个委托类型，它获取两个整数并返回一个字符串。代码清单2-5中的Lambda表达式指出委托实例（在func中）应该求两个整数的乘积，并调用ToString()。该语法比匿名方法的语法简单得多。除此之外，编译器能帮助执行更多的类型推断工作。Lambda表达式无疑是LINQ的关键，你现在就应该准备好把它们变成自己的语言工具包的一个核心部分。然而，它们并非只能用于LINQ，在C# 2中能够使用匿名方法的几乎任何地方都可以在C# 3中使用Lambda表达式，而这样总会让代码更加简短。

总之，和委托有关的新特性包括：

- 泛型（泛型委托类型）——C# 2；
- 创建委托实例时使用的表达式——C# 2；
- 匿名方法——C# 2；
- 委托协变性/逆变性——C# 2；
- Lambda表达式——C# 3。

此外，C# 4还支持在委托中使用泛型的协变和逆变，这已经超出了我们刚刚看到的。确实，泛型是对类型系统进行增强的原则之一，下面将对此进行详述。

2.4.2　与类型系统有关的特性

C# 2中关于类型系统的主要新特性就是泛型。它在很大程度上解决了2.2.2节列出的与强类型的集合有关的问题，虽然泛型类型在其他许多情况下也很有用。这是设计得十分优雅的一个特性，它解决了一个实际的问题。虽然存在少许缺陷，但它在一般情况下都能很好地工作。泛型的例子已出现过多次，而且会在下一章完整地讲述，所以这里不打算给出更多的细节。但这只是暂时的——泛型构成了C# 2类型系统几乎最重要的特性，而且它将贯穿本书剩余部分。

C# 2并没有解决针对覆盖成员和实现接口时返回类型的协变性和参数的逆变性问题。但是，在特定情况下创建委托实例时，C# 2确实使之得到了改善，2.4.1节已对此进行了描述。

C# 3为类型系统引入了许多新概念，最引人注目的是匿名类型、隐式类型的局部变量以及扩

展方法。匿名类型主要是为LINQ而存在的。在LINQ中利用匿名类型，可以有效地创建一个含有大量只读属性的数据转移类型，同时不必真正为它们写代码。然而，在LINQ外部也可以使用匿名类型，它简化了编程。代码清单2-6演示了匿名类型和隐式类型。

代码清单2-6 演示匿名类型和隐式类型

```
var jon = new { Name = "Jon", Age = 31 };
var tom = new { Name = "Tom", Age = 4 };
Console.WriteLine ("{0} is {1}", jon.Name, jon.Age);
Console.WriteLine ("{0} is {1}", tom.Name, tom.Age);
```

前两行代码分别展示了隐式类型（使用var）和匿名对象初始化程序（也就是new {...}部分），后者用于创建匿名类型的实例。

在深入讨论细节之前，重申一下以前曾使我们产生过无谓担心的两件事，这两件事目前一点也不重要。首先，C# 3仍是静态类型的语言。C#编译器将jon和tom声明成一个像普通类型一样的具体的类型。使用对象的属性时，它们是普通的属性——不会发生动态查找（dynamic lookup）。var的意思是说，在声明变量的那一刻，我们（源代码的作者）无法告诉编译器要使用什么类型，编译器到时候会自己生成具体的类型。属性也是静态类型的。在本例中，Age属性是int类型，而Name属性是string类型。

其次，这里没有创建两个不同的匿名类型。变量jon和tom具有相同的类型，因为编译器根据属性名、类型和顺序判断出只需生成一个类型，即可供两个语句使用。该过程是在每个程序集的基础上完成的。这极大简化了编程，因为可以将一个变量的值赋给另一个变量（例如，在上面的代码中，jon=tom;是允许的），以及采取其他类似的操作。

扩展方法同样是为LINQ而生，但在LINQ之外，它也非常有用。以前经常遇到的一个问题是，由于框架类型没有提供一个特定的方法，所以不得不写一个静态的工具方法（utility method）来实现它。例如，通过翻转现有字符串来创建逆序的新字符串，会用到静态的StringUtil.Reverse方法。有了扩展方法之后，就可以直接调用那个静态方法，好像string类型本来就提供了该方法一样。所以可以像下面这样写：

```
string x = "dlrow olleH".Reverse();
```

扩展方法还允许我们表面上为接口添加已经实现了的方法。LINQ在很大程度上依赖这个设计来调用IEnumerable<T>以前根本不存在的各种方法。

C# 4包含两个与类型系统相关的特性。其中相对较小的一个特性是泛型委托和接口的协变与逆变。其实在.NET 2.0时，CLR就已经支持这个特性了，但C#直到第4版（并且在BCL中更新了泛型类型后）才允许我们使用它。另一个大得多的特性是C#的动态类型，不过很多开发者可能永远也不会用到。

还记得吗？我在介绍静态类型时，通过相同的变量调用数组和字符串的Length属性。在C# 4中，这样是可以的。代码清单2-7除了变量的声明外，与之前展示的代码完全相同，可以在C# 4中运行。

代码清单2-7　C# 4中的动态类型

```
dynamic o = "hello";
Console.WriteLine(o.Length);
o = new string[] {"hi", "there"};
Console.WriteLine(o.Length);
```

通过将变量o声明为静态类型dynamic（没错，你看到的是正确的），编译器会对o的几乎所有处理都区别对待，将所有绑定决策（如Length的含义）留给执行时。

我们肯定会对动态类型进行更深入的探讨，但我想在此强调的是，C# 4在很大程度上仍然是一门静态类型的语言。除非使用dynamic类型（表示动态值的静态类型），否则所有的工作都跟以前一模一样。大多数C#开发者都不太需要动态类型，因此可以将它忽略。有了动态类型后，的确是非常方便的，它可以和运行于*动态语言运行时*上的动态语言进行交互。但我建议你不要将C#当成是原生的动态语言来使用。如果你确实需要使用原生的动态语言，可以使用IronPython等专为支持动态类型而设计的语言，它们很少会出现意想不到的错误。

下面简单看一下这些特性，以及各个特性是在C#的哪个版本中引入的：

- *泛型*——C# 2；
- *受限的委托协变性/逆变性*——C# 2；
- *匿名类型*——C# 3；
- *隐式类型*——C# 3；
- *扩展方法*——C# 3；
- *受限的泛型协变/逆变*——C# 4；
- *动态类型*——C# 4。

了解了类型系统在常规意义上的多个不同的新特性之后，接着讨论一下.NET类型系统的一个非常具体的组成部分——值类型——的新特性。

2.4.3　与值类型有关的特性

关于值类型，只有两个特性值得强调，它们都是在C# 2中引入的。第一个特性再次涉及泛型，尤其是“泛型集合”。在.NET 1.1的集合中使用值类型时，人们普遍抱怨的一个问题是，由于所有“常规用途”的API都是用object类型来指定的，所以每次要为集合添加一个结构体值时，都要涉及装箱。获取值时，则又要拆箱。虽然从每次调用来看，装箱的开销并不大，但对于一个需要频繁访问的集合，假如每次都要装箱，就会对性能造成严重影响。另外，由于每个对象的开销，它还使内存的开销无谓增大。泛型使用*真实的*类型，而不只是一个常规用途的对象，从而弥补了在速度和内存使用上的不足。例如，在.NET 1.1中，如果读取一个文件，并将每个字节都存储为ArrayList中的一个元素，那么这无疑是一种“疯狂”之举。但在.NET 2.0中，用List<byte>来做同样的事情，却一点儿都不显得“疯狂”。

有了第二个特性，人们不再抱怨无法将null赋给值类型的变量（尤其是在操作数据库时）。在此之前，像“等于null的int值”这样的概念是不存在的，即使*数据库*中的整数字段允许为空。

所以，很难直接用一个静态类型的类来完美地建模数据库表。可空类型是.NET 2.0的一部分，C# 2添加了额外的语法元素使它们更易使用。代码清单2-8用一个简单的例子进行了演示。

代码清单2-8　演示多种可空类型特性

```
int? x = null;                         <—— 声明并设置可空类型
x = 5;
if (x != null)                         <—— 测试是否存在“真正”的值
{
  int y = x.Value;                     <—— 获取真正的值
  Console.WriteLine (y);
}
int z = x ?? 10;                       <—— 使用null联合操作符
```

代码清单2-8展示了可空类型的大量特性以及C#用于操作它们的简化方式。第4章将详细讨论每个特性，现在最重要的是，你应该思考一下同以前的解决方案相比，所有这些特性有多么容易、多么清晰。

这一次的增强特性列表要简单一些，但无论对于性能的提升还是表达式优雅性的提高，都是十分重要的：

- 泛型——C# 2；
- 可空类型（可以为null的类型）——C# 2。

2.5　小结

本章回顾了C# 1的几个主题。我的目的不是要包括C# 1的方方面面，而只是让大家都站在同一起跑线上，这样才好安心描述更高版本的C#特性，而不必担心读者的基础不够扎实。

前面描述的所有主题其实都是C#和.NET的核心主题。但是，在一些社区讨论中，我经常发现很多人对一些问题存在误解。虽然本章的每个主题都不是特别深入，但应该能帮助你消除一些困惑，从而更容易地理解本书剩余部分的内容。

本章简要讨论的3个核心主题在更高版本的C#中都有了显著增强，而且一些特性涉及多个主题。尤其是泛型，它对本章讨论的几乎所有领域都有影响，它或许是使用最广泛、也是最重要的一个C# 2特性。既然我们已经结束了所有准备工作，下一章就要开始适当地讲述该特性。

Part 2

第二部分

C# 2：解决 C# 1 的问题

第一部分快速浏览了C# 2的几个特性。现在是时候着手真正的工作了。我会描述C# 2如何解决开发者在使用C# 1时遇到的各种问题，以及C# 2如何通过流线（streamlining）使现有的特性变得更有用。这是非常了不起的成就！与C# 1相比，C# 2使我们的编程生活变得舒适和愉快得多。

C# 2的新特性有一定的独立性。当然，这并不是说它们是完全无关的；许多特性都基于泛型对语言巨大的贡献（或者至少与泛型有互动）。然而，接下来的5章讨论的各个主题不能合并成一个整体。

这一部分的前4章包括了以下4个最重要的新特性。

- 泛型——作为C# 2最重要的新特性（同时也是.NET 2.0的CLR中最重要的新特性），泛型实现了类型和方法的参数化[①]。
- 可空类型——值类型（如int和DateTime）没有“值不存在”的概念。有了可空类型之后，就可以表示“缺少一个有意义的值”。
- 委托——虽然委托在CLR的级别上没有任何变化，但C# 2使它们使用起来更容易。除了语法得到了一些简化，匿名方法的引入，还引导我们采取更“函数化”[②]的编程风格——这个趋势在C# 3中得到了延续。
- 迭代器（iterator）——虽然一直以来，都可以利用C#的foreach语句来简单地使用迭代器，但在C# 1中，它实现起来却是一件让人痛苦的事情。C# 2编译器能在幕后帮你构建一个状态机，从而隐藏了大量复杂性。

C# 2每个新的、主要的、复杂的特性都单独用一章来讲述。第7章将讨论几个较简单的特性，从而圆满结束这部分内容。简单并不意味着没用。尤其是分部类型（partial type），它是Visual Studio 2005中提供更好的设计器支持的关键。该特性也有利于其他生成的代码。同样，如今很多开发者都认为能够编写具有公共取值方法和私有赋值方法的属性是理所当然的，但这是C# 2才引入的。

在本书第1版发布时，很多开发者还没有用过C# 2。而在2013年，目前正在使用C#的人中很少有人不熟悉C# 2，可能会不熟悉C# 3，但通常都不熟悉C# 4。我们这里要讨论的主题是C#后续版本的基础。特别是要学习LINQ的话，不理解泛型和迭代器将难以进行。介绍迭代器的那章也涉及C# 5的匿名方法。从表面上看，这两个特性完全不同，但它们都由编译器构建的状态机有关，构建状态机的目的是改变常规的执行流程。

如果你曾经使用过C# 2及其以后的版本，会发现这里的很多内容都很熟悉。但我认为，深入理解了这里所提出的细节，你还是可以从中受益。

① 使什么东西“参数化”，即parameterization，是指那个东西或其成员能作为参数来传递。——译者注
② 所谓“函数化”的编程风格，是指鼓励开发者更多地利用委托。匿名方法和Lambda表达式的引入，使委托变得易于创建和使用。——译者注

第 3 章 用泛型实现参数化类型

本章内容

- ❑ 理解泛型类型和方法
- ❑ 泛型方法的类型推断
- ❑ 类型约束
- ❑ 反射和泛型
- ❑ CLR行为
- ❑ 泛型的限制
- ❑ 与其他语言的对比

一个真实[①]的故事：前几天，我老婆和我准备去超市进行每周例行的购物工作。走之前，她问我是否带了“购物清单”，当我确认带了以后，我们就出发了。但是，等我们到了超市，才发现我俩犯了一个可笑的错误。我老婆要我带的是“购物清单”，而我带的是记录了C# 2简洁新特性的“单子”。当我们问一个店员在哪里能买到“匿名方法”时，他打量我们的眼神奇怪极了。

唉，如果能更清楚地表达自己的意思该有多好！只要我老婆有一个办法能表示她希望我带的是“购物清单”就好了。如果我们有“泛型”……

对于大多数人来说，泛型将成为C# 2最重要的新特性。它们增强了性能，使代码更富有表现力，而且将大量安全检查从执行时转移到了编译时进行。从根本上说，泛型实现了类型和方法的参数化，就像在普通的方法调用中，经常要用参数来告诉它们使用什么值。同样，泛型类型和方法也可以让参数告诉它们使用什么类型。刚开始的时候，一切似乎都不太容易理解。假如你是刚刚接触泛型，对它感到头痛是再正常不过的一件事情。但是，理解了其基本的设计思路后，就会开始对它们爱不释手。

本章将讨论如何使用别人提供的泛型类型和方法（不管是框架中的，还是第三方库中的），以及如何使用泛型来写自己的程序。同时，我们还将学习在API中进行反射调用时泛型是如何工作的，以及CLR处理泛型的一些细节。本章最后将列出泛型最常见的一些限制，以及可能的解决方案，并将C#的泛型与其他语言的类似特性进行了对比。

但是，首先有必要了解因为什么，才导致了泛型的问世。

① 也不必较真，其实我的主要意思是“为了方便引入本章的主题”。

3.1 为什么需要泛型

你手头还有C# 1的代码吗？数一数其中的强制转换有多少——特别是那些大量使用集合的代码。几乎每次使用foreach都需要隐式的强制转换。使用那些为不同的数据类型而设计的类型[①]，就意味着会有强制转换，它们平静地告诉编译器："什么都别担心，一切都很正常，就认为这个表达式具有这种特定的类型就可以了。"任何API只要将object作为参数类型或返回类型使用，就可能在某个时候涉及强制类型转换。设计只有一个类，并将object[②]作为根的层次结构，将使一切变得更加简单。但是，object类型本身是极其"愚钝"的一个存在。要用一个object做真正有意义的事情，几乎都要对它进行强制类型转换。

强制类型转换很糟糕，是吧？它并不是"永远不这样做"的那种糟糕（如易变结构和非私有字段），而是"不得不用"的那种糟糕。但是，如果要想达到某个目的就不得不用，那它就是坏的。发生强制类型转换后，就意味着你本来应该为编译器提供更多的信息，但你选择的是让编译器在编译时相信你，并生成一个检查，以便执行时运行，以验证你所言非虚。

如果需要在某处将某些信息传达给编译器，那么正在读你的代码的人也极有可能需要相同的信息。当然，他们能在你进行强制类型转换时读到这些"信息"，但那几乎没有任何用处。保存这些信息的理想位置通常是声明变量或方法的位置。如果要提供一个其他人不需要访问源代码就能调用的类型或方法，这一点就更加重要。有了泛型，用户在程序中用错误的参数调用库时，就无法通过编译。

在C# 1中，我们需要依赖手工编写的文档，但由于频繁复制信息，因此很难保持完整和正确。如果可以将额外的信息作为方法或类型声明的一部分来加以说明，所有的人都可以更高效地工作：编译器能执行更多的检查；IDE能基于额外的信息向程序员显示"智能感知"选项（例如，访问字符串列表的一个元素时，自动显示string的成员）；方法的调用者对于自己传递的参数值以及方法的返回值可以更有把握；任何人在维护你的代码时，都可以更好地把握你当初写代码时的思路。

说明 **泛型会减少bug数量吗？** 我读到的所有关于泛型的描述（包括我自己的）都强调了编译时类型检查较之执行时类型检查的重要性。但我要与你分享一个秘密：在我已经发布的代码中，记忆中尚未有因为缺乏类型检查而造成过一个bug。换言之，以我的经验来看，C# 1代码中的强制类型转换一直都能很好地工作。那些强制类型转换就像是警告标志，强迫我显式地（主动地）思考类型安全性，而不是在自己的代码中对它们放任自流。有了泛型之后，虽然不一定能从根本上减少与类型安全有关的bug数量，但由于代码变得更易读，所以总体的bug数量就减少了。代码越容易理解，就越不容易写错！同样，在类型系统可以提供适当担保的情况下，编写抵御恶意调用者入侵的健壮代码也容易得多。

以上这些足以证明泛型的存在价值，但还有泛型对性能的增强。首先由于编译器能执行更多的

① 这里指的是ArrayList这样的类型。——译者注

② 作者的意思是指像System.Collections.Queue这样的非泛型的类。在这些类的各个方法中，不仅参数是object类型，返回值也是object类型。——译者注

检查，所以执行时的检查就可以少做。其次，JIT能够聪明地处理值类型，能消除很多情况下的装箱和拆箱处理。某些情况下，无论在速度上还是在内存消耗上，有泛型和没有泛型的结果会大相径庭。

泛型带来的好处非常像静态语言较之动态语言的优点：更好的编译时检查，更多在代码中能直接表现的信息，更多的IDE支持，更好的性能。原因很简单，使用一个不能区分不同类型的常规API（比如`ArrayList`），相当于在一个动态环境中访问那个API。顺便说一下，反过来说通常不成立——动态语言在许多情况下都具备大量的优势，但这些情况很少适用于选择泛型和非泛型的API。当你能合理地使用泛型时，通常都会毫不犹豫地选择泛型。

好了，这些就是C# 2提供的“甜头”，现在到了我们真正开始使用泛型的时候。

3.2　日常使用的简单泛型

搞清楚泛型的方方面面并不是一件容易的事情。围绕这个主题，存在着一些平时不大为人所知的“阴暗角落”。C# 2语言规范提供了丰富的细节，描述了泛型在所有想得到的情况下的行为。但是，为了写出富有效率的程序，并不一定非要去探访每一个这样的“角落”。（同样的道理也适合其他领域。例如，无须知道与“变量确定性赋值”有关的所有细节——等编译器抱怨时，适当地修正一下即可。）

本节包括了平常使用泛型时需要了解的大多数知识（既包括使用别人创建的泛型API，又包括创建你自己的泛型API）。如果在阅读本章时遇到困难，但又急于取得进展，我建议集中精力学习你需要掌握的知识，以便使用.NET Framework和其他库提供的泛型类型和方法。需要自己写泛型类型和方法的时候并不多，一般都是直接使用框架提供的。

先来看看.NET 2.0提供的一个集合类：`Dictionary<TKey,TValue>`。

3.2.1　通过例子来学习：泛型字典

只要不是凑巧撞到了一些限制，然后开始想哪里出错了，使用泛型类型还是相当简单的。不需要知道任何术语，直接读代码即可大致猜出它要做的事情。经过少许尝试和犯错之后，就可以摸索出写一些可运行的代码的方法。（泛型的好处之一就是在编译时执行更多的检查，所以等到编译不再报错时，就极有可能已经得到了能正常工作的代码——这使试验变得更简单。）当然，本章的目的是教给你知识，使你不必再猜测——在每个阶段都能把握事态的发展。

但是现在，先来看看一些简单的代码，即使不熟悉语法也没关系。代码清单3-1使用一个`Dictionary<TKey,TValue>`（大致相当于你在C# 1中肯定用过的`Hashtable`类）来统计单词在一段给定文本中的出现频率 。

代码清单3-1　用`Dictionary<TKey,TValue>`统计文本中的单词数

```
static Dictionary<string,int> CountWords(string text)
{
   Dictionary<string,int> frequencies;          ❶ 创建从单词到频率的新映射
   frequencies = new Dictionary<string,int>();

   string[] words = Regex.Split(text, @"\W+");  ❷ 将文本分解成单词
```

```
    foreach (string word in words)
    {
        if (frequencies.ContainsKey(word))
        {
            frequencies[word]++;
        }

        else
        {
            frequencies[word] = 1;
        }
    }
    return frequencies;
}
...
string text = @"Do you like green eggs and ham?
                I do not like them, Sam-I-am.
                I do not like green eggs and ham.";

Dictionary<string,int> frequencies = CountWords(text);
foreach (KeyValuePair<string,int> entry in frequencies)
{
    string word = entry.Key;
    int frequency = entry.Value;
    Console.WriteLine ("{0}: {1}", word, frequency);
}
```

❸ 添加或更新映射

❹ 打印映射中的每个键/值对

CountWords方法❶首先创建从string到int的一个空白映射。它将有效地统计每个单词在一段给定文本中出现的频率。然后，用一个正则表达式❷将文本分解成单词。这是非常粗糙的一种处理方式——最终会得到两个空字符串（文本头尾各一个），没有去管“do”和“Do”被分开计数的问题[①]。这些问题很容易修正，这个例子只是想使代码尽可能简单。

对于每个单词，都检查它是否已经在映射中。如果是，就增加现有计数；否则，就为单词赋予一个初始计数1❸。注意，负责递增的代码不需要执行到int的强制类型转换，就可以执行加法运算：取回的值在编译时已知是int类型。使计数递增的步骤实际是先对映射的索引器执行一次取值操作，然后增加，然后对索引器执行赋值操作。有的开发者可能觉得显式表达更清晰一些，就换成frequencies[word] = frequencies[word]+1;。

上述代码清单的最后一部分应该是非常熟悉的：遍历一个Hashtable，这个Hashtable包含相似的（非泛型）DictionaryEntry，其中每个条目都具有Key和Value属性❹。但在C# 1中，word和frequency都需要进行强制类型转换，因为Key和Value都返回object类型。另外，frequency还需要进行装箱。显然，这里并非一定要定义word和frequency变量，只需直接调用Console.WriteLine，并将entry.Key和entry.Value作为参数传递就可以了。之所以用了变量，纯粹是为了证明没有必要使用强制类型转换。

在介绍了一个例子之后，我们先来看看Dictionary<TKey,TValue>的真正含义。什么是

① 实际上，这里除了单词，还会得到一个空字符串——这是“green eggs and ham.”末尾的句点符号造成的。正则表达式看到一个非单词字符，所以会将“ham.”分解为“ham”和“.”。——译者注

TKey和TValue，为什么要用尖括号把它们括起来？

3.2.2　泛型类型和类型参数

泛型有两种形式：*泛型类型*（包括类、接口、委托和结构——没有泛型枚举）和*泛型方法*。两者都是表示API的基本方法（不管是指单独的泛型方法还是完整的泛型类型），在平时期望出现一个普通类型的地方，用一个类型参数。

类型参数是真实类型的占位符。在泛型声明中，类型参数要放在一对尖括号内，并以逗号分隔。所以，在Dictionary <TKey,TValue>中，类型参数是TKey和TValue。使用泛型类型或方法时，要用真实的类型代替。这些真实的类型称为*类型实参*（type argument）。在代码清单3-1中，类型实参是string（代替TKey）和int（代替TValue）。

说明　专业术语的变化！　讨论泛型时会涉及大量专业术语。我在这里也使用了这些术语，供大家参考。另外，在极少数情况下，用这些术语可以更准确地阐述一个主题。如果需要查阅语言规范，知道这些术语也是很有帮助的。但是，平常[①]一般不必使用这些术语。如果觉得别扭，请暂时容忍一下。C# 5语言规范的4.4节“已构造类型”中定义了许多这样的术语，可以参考。

如果没有为泛型类型参数提供类型实参，那么这就是一个*未绑定泛型类型*（unbound generic type）。如果指定了类型实参，该类型就称为一个*已构造类型*（constructed type）。我们知道，类型（无论是否是泛型）可以看做是对象的蓝图。同样，未绑定泛型类型是已构造类型的蓝图。它是一种额外的抽象层[②]。图3-1对此进行了展示。

更复杂的是，已构造类型可以是开放或封闭的。*开放类型*（open type）还包含一个类型参数（例如，作为类型实参之一或数组元素类型），而*封闭类型*（closed type）则不是开放的，类型的每个部分都是明确的。所有代码实际都是在一个封闭的已构造类型的上下文中执行。在C#代码中，唯一能看见未绑定泛型类型的地方（除了作为声明之外）就是在typeof操作符内。3.4.4节将讨论typeof操作符。

类型参数“接收”信息，类型实参“提供”信息（图3-1上半部分的3条虚线），这个思路与方法参数和方法实参是一样的。只不过类型实参必须为类型，而不能为任意的值。只有在编译时才能知道类型实参的类型，它可以是（或包含）相关上下文中的类型参数。

---

① 平时很少需要区分形参和实参，也就是parameter和argument。一般都把它们称为“参数”。需要区分的时候，本书将parameter译为“参数”，将argument译为“实参”。我使用“类型参数”时，实际是指“类型形参”。另外，在C#语言规范中，使用的也是作者使用的这些术语。为便于参考，术语初次出现时会列出原文。——译者注

② 未绑定泛型类型是已构造类型的蓝图，已构造类型是实际的对象的蓝图，正是因为存在这个关系，所以才有“额外的抽象层”一说。——译者注

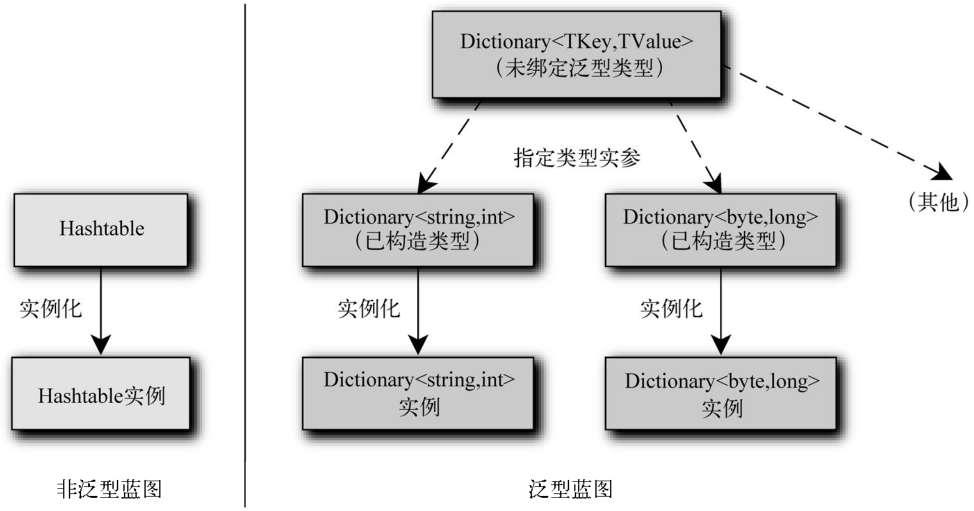


图3-1　未绑定泛型类型相当于已构造类型的蓝图。已构造类型又是实际对象的蓝图，这一点和非泛型类型的作用是相似的

可以认为“封闭类型”拥有“开放类型”的API，只不过类型参数被替换成了对应的类型实参[①]。表3-1展示了开放类型`Dictionary<TKey,TValue>`的一些方法和属性声明，以及根据它来构造的封闭类型`Dictionary<string,int>`中的等价成员。

表3-1　在泛型类型中的方法签名中包含占位符的方式，这些占位符会在指定类型实参时被替换

泛型类型中的方法签名	类型参数被替换之后的方法签名
`void Add(TKey key, TValue value)`	`void Add(string key, int value)`
`TValue this[TKey key]{ get; set; }`	`int this[string key]{ get; set; }`
`bool ContainsValue(TValue value)`	`bool ContainsValue(int value)`
`bool ContainsKey(TKey key)`	`bool ContainsKey(string key)`

要注意的一个重点是，在表3-1中，没有任何一个方法是真正的泛型方法。它们只是泛型类型中的“普通”方法，只是凑巧使用了作为类型一部分声明的类型参数。下一节，我们会讲述泛型方法。

你已经知道了`TKey`和`TValue`的含义，并知道了那些尖括号有什么用，接着来看看表3-1中的内容在类中是如何声明的。以下是`Dictionary<TKey, TValue>`可能的代码，虽然没有包括任何实际的方法实现，而且实际的成员数量比这更多：

① 其实情况并不总是这样，有一些边缘情况不符合这个简单的规则，但可以认为规范在大多数情况下是这么工作的。——译者注

```
namespace System.Collections.Generic
{
   public class Dictionary<TKey,TValue>          ← 声明泛型类型
      : IEnumerable<KeyValuePair<TKey,TValue>>    ← 实现泛型接口
   {
      public Dictionary() { ... }                 ← 声明无参构造函数

      public void Add(TKey key, TValue value) { ... }   ← 使用类型参数声明方法

      public TValue this[TKey key]
      {
         get { ... }
         set { ... }
      }

      public bool ContainsValue(TValue value) { ... }

      public bool ContainsKey(TKey key) { ... }

      [... other members ...]
   }
}
```

注意Dictionary<TKey,TValue>是如何实现泛型接口IEnumerable<KeyValuePair<TKey,TValue>>的(事实上还实现了其他许多接口)。为类指定的任何类型实参都可以应用到具有相同类型参数的接口中。所以，在我们的例子中，Dictionary<string,int>会实现IEnumerable<KeyValuePair<string,int>>。后者实际是某种“双重泛型”接口——它是一个IEnumerable<T>接口，其类型实参是KeyValuePair<string,int>结构。正是由于实现了IEnumerable<KeyValuePair<string,int>>接口，所以代码清单3-1最后才能像那样枚举keys和values。

还值得指出的是，构造函数不在尖括号中列出类型参数。类型参数从属于类型，而非从属于某个特定的构造函数，所以才会在声明类型时声明。成员（仅限方法）仅在引入新的类型参数时才需要声明。

说明　***泛型的发音***　在向其他人描述泛型类型时，通常使用of来介绍类型参数或实参，因此List<T>读作list of T。在VB中，of是语言的一部分，泛型类型本身就写作List(Of T)。当有多个类型参数时，可以用一个适合整个类型含意的单词来分隔它们，例如，我会使用dictionary of string to int来强调映射的部分，而不对tuple使用of string and int。

泛型类型可以重载，只需改变一下类型参数的数量就可以了。例如，可以定义MyType、MyType<T>、MyType<T,U>、MyType<T,U,V>……以此类推。所有这些定义都可以放到同一个命名空间内。类型参数的名称并不重要，重要的是个数。这些类型除了名称相同，别的没什么联系（例如，不存在从一个类型到另一个类型的默认转换）。这一点对泛型方法也是成立的：除了类型参数的数量，两个方法的签名可以完全相同。尽管这听上去似乎是场灾难，但如果想充分利用泛型类型推断，它十分有用，编译器可以为我们计算出类型实参。3.3.2节将再次讨论这些内容。

说明 **类型参数命名规范** 虽然可以使用带有T、U和V这样的类型参数的类型，但从中根本看不出实际指的是什么，也看不出它们应该如何使用。相比之下，像Dictionary<TKey, TValue>这样的命名就要好得多，TKey明显代表键的类型，而TValue代表值的类型。如果只有一个类型参数，而且它的含义很清楚，那么一般使用T（List<T>就是一个很好的例子）。如果有多个类型参数，则应根据含义来命名，并用T前缀来指明这是一个类型参数。虽然有时可能会遇到一个类型带有多个单字母的类型参数（比如Synchronized-KeyedCollection <K,T>），但应该避免自己创建这样的类型。

在了解了泛型所做的事情之后，接着来研究一下泛型方法。

3.2.3 泛型方法和判读泛型声明

以前已提到过几次泛型方法，只是还没有真正遇到过这样的方法。泛型方法的概念或许比泛型类型更容易让人“犯迷糊”，它们不太符合我们的惯常思维，但基本原理是相同的。我们已习惯于方法的参数和返回值有固定的类型，而且已看到了泛型类型如何在它的方法声明中使用类型参数[①]。泛型方法则更进一步，即使你已经确切地知道要操作哪一个已构造类型，泛型方法也可以有类型参数。不懂这句话也没有关系——见过足够多的例子之后，就会豁然开朗。

虽然Dictionary<TKey,TValue>没有任何泛型方法，但它的近亲List<T>是有的。你能猜得到，List<T>表示的是由任意指定类型的数据项构成的列表。例如，List<string>是一个字符串列表。记住：T是在整个类的范围内使用的类型参数。然后，让我们分析一个泛型方法的声明。图3-2展示了ConvertAll方法声明的各个部分的含义。

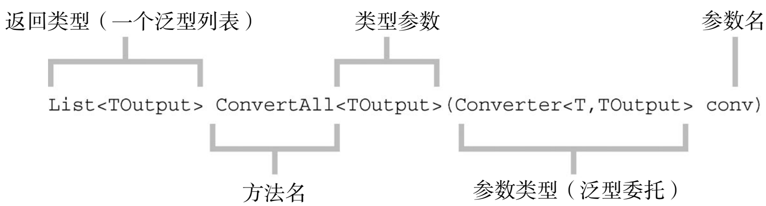


图3-2 泛型方法声明剖析

看一个泛型声明时，无论它声明的是一个泛型类型还是一个泛型方法，要分析出它的真正含义总是有点让人气馁，尤其是在处理“泛型类型的泛型类型”问题时，就像前面见过的由字典实现的接口一样。这时的要诀是不要惊慌——静下心来，然后选择同类型的例子。为每个类型参数使用一个不同的类型，再整体应用这些类型参数。

① 作者的意思是在泛型类型自己的方法中使用类型已经声明的类型参数，但这些方法绝不是泛型方法。参见表3-1。——译者注

在本例中，首先替换包含方法（List<T>的<T>部分）的那个类型的类型参数。在本例中，首先替换该方法所属类型的类型参数（List<T>的<T>部分）。我们仍然使用字符串列表，将方法声明中所有的T替换为string：

```
List<TOutput> ConvertAll<TOutput>(Converter<string,TOutput> converter)
```

现在看起来要好一些了，但还有TOutput需要处理。可以判断出它是一个方法的类型参数（我要为混淆的术语道歉），因为它位于紧跟在方法名后面的尖括号中。所以，尝试用另一个熟悉的类型Guid作为TOutput的类型实参。我们再次用类型实参替换所有的类型参数。现在可以移除声明中的类型参数，将该方法看成是非泛型的：

```
List<Guid> ConvertAll(Converter<string,Guid> converter)
```

现在，所有的内容都由具体的类型表示，看上去简单一些了。不过真正的方法仍然为泛型的，我们只是为了便于理解而将它看成是非泛型的。现在来从左向右分析声明中的各个元素：

- 方法返回一个List<Guid>；
- 方法名是ConvertAll；
- 方法有一个参数，该参数是名为converter的一个Converter<string,Guid>。

现在，只需知道Converter<string,Guid>是什么，一切就都清楚了。不足为奇，Converter<string,Guid>是一个已构造的*泛型委托类型*（未绑定类型是Converter<TInput,TOutput>），可以将字符串转换成GUID。

这样一来，我们就知道该方法能处理一个字符串列表，用一个转换器来生成一个GUID列表。在理解了方法的签名之后，就很容易理解文档。在文档中，确认这个方法能做一些显而易见的事情以及创建新的List<Guid>——将原始列表中的每个元素都转换成目标类型，将转换后的元素添加到一个列表中，最后返回这个列表。对抽象的方法签名进行具体化，有助于理清思路，而且可以更轻松地理解方法要做的事情。尽管这项技术听上去有点简单，但即使现在它对于那些复杂的方法来说也是有用的。一些LINQ方法的签名包含4个类型参数，这听上去非常可怕，但转换为具体的名词可以显著地简化它们。

为了证明我没有误导你，下面实际使用一下这个方法。代码清单3-2将一个整数列表转换成浮点数列表，第二个列表的每个元素都是第一个列表中对应元素的平方根。转换之后，输出结果。

代码清单3-2　List<T>.ConvertAll<TOutput>方法实战

```
static double TakeSquareRoot(int x)
{
   return Math.Sqrt(x);
}
...
List<int> integers = new List<int>();          ❶ 创建并填充一个整数列表
integers.Add(1);
integers.Add(2);
integers.Add(3);
integers.Add(4);
```

```
Converter<int,double> converter = TakeSquareRoot;   ❷ 创建委托实例
List<double> doubles;
doubles = integers.ConvertAll<double>(converter);   ❸ 调用泛型方法来转换列表
foreach (double d in doubles)
{
   Console.WriteLine(d);
}
```


列表的创建和填充过程❶是很简单的——这只是一个强类型的整数列表。converter的赋值❷使用了委托的一个特性（方法组转换），这是C# 2新增的，将在5.2节更详细地讨论。虽然我不喜欢在完整地描述一个特性之前使用它，但假如使用完整版本，这一行就会变得过长，以致超出这页的范围。不过，它确实能做你希望它做的事情。在❸中，我们调用泛型方法。调用时，使用为泛型类型指定类型实参的方式来为泛型方法指定类型实参。这里可以使用类型推断来避免显式指定类型实参，但我希望一步一步来。最后，可以非常简单地写出返回的列表内容。运行代码，会看到它打输出1，1.414…，1.732…和2，这和我们希望的结果是一样的。

那么，这一切到底是为了说明什么问题？完全可以用一个foreach循环来遍历所有整数，并立即打印每个整数的平方根。但是，对一种类型的列表进行一番逻辑处理，把它转换成另一种类型的列表，这是十分常见的一种操作。虽然手动完成这个过程的代码仍然十分简单，但假如在一个方法调用中就能完成，那么显然更易读。这正是泛型方法的价值所在——以前很得意地用“普通方法”来完成的事情，现在只需一个方法调用就可完成。在引入泛型以前，可以使用与ConvertAll类似的方式对ArrayList进行操作，从而将object转换成object，但效果显然不如现在这样令人满意。匿名方法（参见5.4节）在这里也可以提供帮助——如果不想引入一个额外的方法，那么可以采取内联的方式来指定要进行的转换。LINQ和Lambda表达式大量使用了这种模式，第三部分将进行介绍。

注意，非泛型类型也可以拥有泛型方法。代码清单3-3展示了在一个普通的类中声明和使用的泛型方法。

代码清单3-3　在非泛型类型中实现泛型方法

```
static List<T> MakeList<T>(T first, T second)
{
   List<T> list = new List<T>();
   list.Add(first);
   list.Add(second);
   return list;
}
...
List<string> list = MakeList<string>("Line 1", "Line 2");
foreach (string x in list)
{
   Console.WriteLine (x);
}
```

MakeList<T>泛型方法只需一个类型参数（T）。它只是创建一个包含两个参数的列表。然而，在方法中创建List<T>时，可以使用T作为类型实参。和学习泛型声明时一样，可以将方法的实现看做是将所有T替换成string。调用方法时，使用和前面见到的一样的语法来指定参数类型。

一切都还好吧？现在，你应该已经掌握了“简单”泛型。我担心，后面的内容可能会复杂一些。但是，只要真正理解了泛型的基本概念，那么最大的障碍其实已经跨过去了。如果对此仍然有点儿模糊，尤其是在遇到开放/封闭/未绑定/已构造等术语时，也不要过于担心。不过，现在是亲自进行试验的好时机。可以在学习后面的内容之前，对泛型进行实战演练。如果你以前没用过泛型集合，可以快速浏览一下附录B，它描述了所有可用的泛型集合。这些集合类型既可以作为你初学泛型的起点，也可以在几乎所有大型的.NET程序中广泛使用。

进行试验时，你可能发现自己很难做到“浅尝辄止”。将API的一部分变为泛型以后，极有可能还需要重写其他代码，要么把它们也变成泛型，要么根据现在强类型方法调用的需要进行强制类型转换。一个可选的方案是使用强类型的实现，在底下使用泛型类，但暂时保留一个弱类型的API。随着时间的推移，你会越来越准确地把握使用泛型的时机。

3.3　深化与提高

虽然前面介绍的泛型的简单用法已经使你取得了很大进步，但还有更多的特性可供利用，它们能提供进一步的帮助。

本节首先讨论的是*类型约束*（type constraint），它用于进一步控制指定哪一个类型实参。创建你自己的泛型类型和方法时，类型约束是很有用的。另外，只有理解了类型约束，使用框架时才知道可以使用哪些选项。

其次要讨论的是*类型推断*（type inference）——这是编译器变的一个“戏法”。它意味着在使用泛型方法时，并非一定要显式地声明类型实参。虽然不用它也行，但假如使用恰当，可以使代码变得更易读。第三部分会讲到，C#编译器能逐渐地从你的代码中推断出越来越多的信息，同时仍能保持语言的安全性和静态类型[①]。

本节的最后一部分要讨论如何获取类型参数的默认值，以及在写泛型代码时可以进行哪些比较。最后我们将用一个例子来演示这些特性，这个例子还提供了一个非常有用的类。

虽然本节在泛型主题上更深入了一些，但并没有特别难的地方。虽然要记住的知识点很多，但所有特性都服务于一个目的。另外，等以后需要用到这些特性时，就会深刻地体会到它们的好处，我们开始吧。

3.3.1　类型约束

到目前为止，我们见过的所有类型参数都可以指定为任意类型，它们*未被约束*。如一个`List<int>`，`<object,FileMode>`等。处理集合时，如果不需要同其内部存储的东西进行交互，这样做就没有问题。但是，泛型并非都是这么用的。我们经常需要调用类型参数实例的方法，创建新实例，或者想确保只接受引用类型（或者只接受值类型）。换言之，我们想制定规则，从而判断哪些是泛型类型或泛型方法能接受的有效类型实参。在C# 2中，这用*约束*（constraint）

① C# 4中的代码可以显式地使用动态类型，这是例外。

来实现。

有4种约束可供使用，它们的常规语法相同。约束要放到泛型方法或泛型类型声明的末尾，并由上下文关键字where来引入。稍后会看到，它们能按合理的方式组合到一起。但在此之前，先来讲讲每一种约束。

1. 引用类型约束

第一种约束用于确保使用的类型实参是引用类型（它表示成T : class，且必须是为类型参数指定的第一个约束）。类型实参任何类、接口、数组、委托，或者已知是引用类型的另一个类型参数。例如以下声明：

```
struct RefSample<T> where T : class
```

有效的封闭类型包括：

- RefSample<IDisposable>;
- RefSample<string>;
- RefSample<int[]>。

无效的封闭类型包括：

- RefSample<Guid>;
- RefSample<int>。

我故意将RefSample声明为一个struct（所以类型本身仍然是值类型），以强调被约束的类型参数和类型本身的差异。RefSample<string>仍是值类型，处处都是值的语义——只是在API中指定T的地方要使用string类型。

以这种方式约束了一个类型参数后，可以使用==和!=来比较引用（包括null）。但要注意的是，除非还存在其他约束，否则只能比较引用，即使该类型中重载了那些操作符（例如string就是这样）。使用一个转换类型约束（稍后会讲述），可以由编译器保证对==和!=进行重载。在这种情况下就会使用重载的版本，只是这种情况极其罕见。

2. 值类型约束

这种约束表示成T : struct，可以确保使用的类型实参是值类型，包括枚举（enums）。但是，它将可空类型（将在第4章讲述）排除在外。来看一个示例声明：

```
class ValSample<T> where T : struct
```

有效的封闭类型包括：

- ValSample<int>;
- ValSample<FileMode>。

无效的封闭类型包括：

- ValSample<object>;
- ValSample<StringBuilder>。

这一次，ValSample类型本身是一个引用类型，虽然T被约束成值类型。注意，System.Enum和System.ValueType本身都是引用类型，所以不允许作为ValSample的类型实参使用。类型

参数被约束为值类型后，就不允许使用==和!=进行比较。

我很少有使用值类型或引用类型约束，虽然下一章会看到可空类型要依赖于值类型约束。剩余的两种约束在你自己写泛型类型时可以发挥更大的作用。

3. 构造函数类型约束

构造函数类型约束表示成T : new()，必须是所有类型参数的最后一个约束，它检查类型实参是否有一个可用于创建类型实例的无参构造函数。这适用于所有值类型；所有没有显式声明构造函数的非静态、非抽象类；所有显式声明了一个公共无参构造函数的非抽象类。

说明　C#与CLI标准　涉及值类型和构造函数时，C#和CLI标准有一个不一致的地方。C#规范则规定，所有值类型都有一个默认的无参构造函数，而且显式声明的构造函数和无参构造函数是用相同的语法来调用的。至于具体调用哪一个，要依赖于编译器正在底层进行的工作。CLI规范则没有这些要求，不过它提供了一个特殊的指令，可以在不指定任何参数的情况下创建默认值。用反射来找出一个值类型的构造函数时，可以真正看到这个差异——你看不到无参构造函数。

同样，让我们用一个简单的例子加以说明。这次的例子是一个泛型方法。为了演示它的用法，还给出了具体的实现：

```
public T CreateInstance<T>() where T : new()
{
   return new T();
}
```

假如由你指定的类型有一个无参构造函数，这个方法将返回该类型的一个新实例。所以，CreateInstance<int>()和CreateInstance<object>()都是有效的。但是，CreateInstance<string>();是无效的，因为string没有无参构造函数。

没有办法规定类型参数必须具备其他构造函数签名。例如，无法指定类型参数必须拥有一个以单个字符串作为参数的构造函数。这令人沮丧，但遗憾的是，它就是这样的。3.5节探讨.NET泛型的各种约束时，会详细介绍这个问题。

使用工厂风格的设计模式时，构造函数类型约束非常有用。在这种设计模式中，一个对象将在需要时创建另一个对象。当然，工厂经常需要生成与一个特定接口兼容的对象——这就引入了最后一种约束。

4. 转换类型约束

最后（也是最复杂的）一种约束允许你指定另一个类型，类型实参必须可以通过一致性、引用或装箱转换隐式地转换为该类型。你还可以规定一个类型实参必须可以转换为另一个类型实参——这称为*类型参数约束*（type parameter constraint）；虽然这会使声明变得更难以理解，但有时候还是有用的。表3-2演示了一些带有转换类型约束的泛型类型声明，并分别列举了有效和无效的已构造类型。

表3-2 转换类型约束的例子

声 明	已构造类型的例子
`class Sample<T> where T : Stream`	有效：`Sample<Stream>`（一致性转换） 无效：`Sample<string>`
`struct Sample<T> where T : IDisposable`	有效：`Sample<SqlConnection>`（引用转换） 无效：`Sample<StringBuilder>`
`class Sample<T> where T : IComparable<T>`	有效：`Sample<int>`（装箱转换） 无效：`Sample<FileInfo>`
`class Sample<T,U> where T : U`	有效：`Sample<Stream,IDisposable>`（引用转换） 无效：`Sample<string,IDisposable>`

第3个约束`T : IComparable<T>`将泛型类型作为约束。其他形式，比如`T : List<U>`（`U`是另一个类型参数）和`T : IList<string>`也是正确的。

可以指定多个接口，但只能指定一个类。例如，以下声明毫无问题（尽管很难满足）：

```
class Sample<T> where T : Stream,
                          IEnumerable<string>,
                          IComparable<int>
```

但以下声明就有问题了：

```
class Sample<T> where T : Stream,
                          ArrayList,
                          IComparable<int>
```

总之，任何类型都不能直接派生自多个类。对于这样的一个约束，要么是永远无法满足的（就像上例那样），要么它的一部分是多余的（例如，规定类型必须从`Stream`和`MemoryStream`派生）。

此外，还有一系列限制：指定的类不可以是结构、密封类（比如`string`）或者以下任何“特殊”类型：

- `System.Object`；
- `System.Enum`；
- `System.ValueType`；
- `System.Delegate`。

说明 **围绕缺乏枚举和委托约束所做的工作** 在转换类型约束中无法指定前面提到的这些类型，这听上去似乎是CLR的限制，其实不然。它似乎历史悠久（从设计泛型的时候开始），但实际上如果在IL中构造适当的代码，是可以工作的。CLI规范甚至还举例列出了枚举和委托约束，并解释哪些是有效的，哪些是无效的。这很令人沮丧，很多泛型方法的类型参数如果约束为委托或枚举，将是非常有用的。我开发了一个开源项目Unconstrained Melody（http://code.google.com/p/unconstrained-melody/），它执行了一些黑客[①]方法构建了一个类库，可以对各种工具方法声明这些约束。尽管C#编译器不允许你声明这些约束，但可以在调用类库中的方法时使用它们。也许在C#以后的版本中，这种限制将被取消。

① 这里使用的是hacker的本意，即对计算机科学、编程和设计方面具有高度理解的人。——译者注

转换类型约束也许是最有用的一种约束，因为它们意味着可以“在类型参数的实例上使用指定类型的成员”。最典型的例子是T : IComparable<T>，它意味着能直接地、有意义地对T的两个实例进行比较。3.3.3节会讲述一个这方面的例子，并讨论其他比较形式。

5. 组合约束

前面已提到了应用多个约束的可能性，而且在转换类型约束中，已见到过它们。但是，我们还没有看到过将不同种类的约束合并到一起的情况。显然，没有任何类型既是引用类型，又是值类型。所以，像这样的组合是禁止的。另外，由于每一个值类型都有一个无参构造函数，所以假如已经有一个值类型约束，就不允许再指定一个构造函数约束（但是，如果T被约束成一个值类型，仍然可以在方法内部使用new T()）。如果存在多个转换类型约束，并且其中一个为类，那么它应该出现在接口的前面，而且我们不能多次指定同一个接口。不同的类型参数可以有不同的约束，它们分别由一个单独的where引入。

来看一些有效和无效的例子。

有效：

```
class Sample<T> where T : class, IDisposable, new()
class Sample<T> where T : struct, IDisposable
class Sample<T,U> where T : class where U : struct, T
class Sample<T,U> where T : Stream where U : IDisposable
```

无效：

```
class Sample<T> where T : class, struct
class Sample<T> where T : Stream, class
class Sample<T> where T : new(), Stream
class Sample<T> where T : IDisposable, Stream
class Sample<T> where T : XmlReader, IComparable, IComparable
class Sample<T,U> where T : struct where U : class, T
class Sample<T,U> where T : Stream, U : IDisposable
```

在两个列表的最后，我使用了相同的例子，这是因为很容易将有效的语句写成无效的，而且编译器的报错没有任何帮助。记住，每个类型参数的约束列表都要单独用一个where引入。第3个“有效”例子比较有趣，如果U是值类型，它是如何从引用类型T派生的呢？答案是T可能是object，或者是U实现的一个接口。但是，这个约束不怎么好。

说明 规范中的术语 规范中对约束的分类略有不同，它将其划分为主要约束、次要约束和构造函数约束[①]。主要约束可以为引用类型约束、值类型约束或使用类的转换类型约束。次要约束为使用接口或其他类型参数的转换类型约束。我不认为这种分类特别有用，不过它们可以使约束语法的定义变得简单：主要约束是可选的，但只能有一个；次要约束则可以有多个；构造函数约束也是可选的（如果拥有了值类型约束，就不能再使用构造函数约束）。

---

① 参见规范的10.1.5节。——译者注

你已经掌握了阅读泛型类型声明所需的一切知识，下面开始讨论前面提到过的类型实参推断。在代码清单3-2中，我们为List.ConvertAll显式声明类型实参。同样，在代码清单3-3中，为我们自己的方法MakeList显式声明类型实参。现在，要让编译器自己判断是什么类型，从而简化泛型方法的调用。

3.3.2 泛型方法类型实参的类型推断

调用泛型方法时，指定类型实参常常会显得很多余。根据方法本身的实参，很容易看出类型实参应该是什么。为了简化工作，C# 2编译器被赋予了一定的“智能”（以严格定义的方式），让你在调用方法时，不需要显式声明类型实参。在深入讨论这个主题之前，必须强调一下：类型推断只适用于泛型*方法*，不适用于泛型*类型*。

澄清了这一点之后，再看看如何简化代码清单3-3中相关的行。下面是声明并调用方法的几行代码：

```
static List<T> MakeList<T>(T first, T second)
...
List<string> list = MakeList<string>("Line 1", "Line 2");
```

现在来看一下我们指定的实参——都是字符串。方法中的每个参数都声明为类型T。即使拿掉方法调用表达式中的<string>部分，也很容易看出在调用方法时，为T使用的类型实参是string。编译器允许将其省略，变成：

```
List<string> list = MakeList("Line 1", "Line 2");
```

是不是要简洁一些？至少这行代码变短了。当然，这并非总是意味着增加了可读性。某些情况下，它甚至会让读者更难判断你要使用什么类型实参——即使编译器能轻松地判断出来。我建议你具体情况具体分析。推断可用时，*大多数*情况下我个人倾向于让编译器推断类型实参。

注意，编译器之所以能够确定我们要将string作为类型实参使用，是因为对list的赋值也在起作用，它也指定了（并且必须指定）类型实参[①]。然而，这种赋值并不会影响类型参数推断过程，它只是意味着假如编译器推断错了你想要使用的类型实参，你仍可能得到一个编译时错误。

编译器为什么会弄错呢？假定实际想要使用object作为类型实参。我们传递的方法实参仍是有效的，但编译器认为我们实际是要使用string，因为传递的两个实参都是字符串。修改其中一个实参，把它显式转换成object，类型推断就会失败，因为一个方法实参指出T应该是string，另一个指出T应该是object。此时，编译器会考虑这种情况并且告知用户将T设为object能满足一切条件，将T设为string则不能。但是，在C#语言规范中，只提供了数量有限的推断步骤。这个主题在C# 2中就已经很复杂了，在C# 3中更甚。我不打算逐条剖析C# 2的所有规则，其基本步骤如下。

(1) 对于每一个方法实参（普通圆括号中的参数，而不是尖括号中的），都尝试用十分简单的技术推断出泛型方法的一些类型实参。

① 这句话是呼应作者在前面强调的一点：类型推断只适用于泛型方法，不适用于泛型类型。——译者注

(2) 验证步骤(1)的所有结果都是一致的——换言之，假如从一个方法实参推断出了某类型参数的类型实参，但根据另一个方法实参推断出同一个类型参数具有另一个类型实参，则此次方法调用的推断失败。

(3) 验证泛型方法需要的所有类型实参都已被推断出来。不能让编译器推断一部分，自己显式指定另一部分。要么全部推断，要么全部显式指定。

为了避免学习所有推断规则（我也不建议那样做，除非你对细节非常感兴趣），只需做一件简单的事情：亲自试一试，看看会发生什么。如果认为编译器也许能推断出所有类型实参，就尝试直接调用方法，不要指定任何类型参数。如果调用失败，就显式指定类型实参。除了要花点时间让编译器编译一次代码之外，其他什么都不会损失，并且你不必记忆乱七八糟的语言规则。

为了简化泛型类型的使用，类型推断可以根据类型参数的数量，识别重载方法中的类型名称。稍后进行总结时，将给出这样的示例。

3.3.3　实现泛型

虽然以后可能大多数时间都是在使用而不是编写泛型类型和方法，但需要由你提供实现时，应该知道几件事情。大多数时候，都可以假装`T`（或者任意类型参数的名称）是一个真正的类型名称，并像平时那样写代码，好像自己根本没在使用泛型。但是，有几件额外的事情需要注意。

1. 默认值表达式

如果已经明确了要处理的类型，也就知道了它的“默认”值，例如未初始化字段的默认值。不知道要引用的类型，就不能直接指定默认值。你不能使用`null`，因为它可能不是一个引用类型。也不能使用0，因为它可能不是数值类型。

虽然很少需要用到默认值，但它偶尔还是有用的。`Dictionary<TKey, TValue>`就是一个很好的例子——它有一个`TryGetValue`方法，它的作用有点儿像对数值类型进行处理的`TryParse`方法：它用一个输出参数来接收你打算获取的值，用一个`Boolean`返回值显示它是否成功。这意味着方法必须用`TValue`类型的值来填充输出参数。（输出参数必须在方法正常返回之前赋值）。

说明　TryXXX模式　.NET有几个模式很容易根据所涉及的方法名称来识别。例如，`BeginXXX`和`EndXXX`暗示着一个异步操作。`TryXXX`模式的用途在从.NET 1.1升级到2.0期间进行了扩展。它是针对以下情况设计的：有些错误虽然一般会被视为错误（在这种情况下，方法不能履行其基本职责），但并不是什么严重的问题，也不应该视为异常。例如，用户在键入数字时很容易出错，所以假如能先尝试解析一下文本，但又不必捕捉并处理异常，那么肯定是相当有用的。这不仅能在容易出错的情形中提升性能，更重要的，它只在处理真正的错误——也就是系统中的问题（可以将问题解释得非常透彻）时才捕获并处理异常。这是库设计者必备的一个模式；如果使用得当，能发挥巨大的作用。

为了满足这方面的需求，C# 2提供了默认值表达式（default value expression）。虽然C#语言规范没有说它是一个操作符，但可以把它看做是与`typeof`相似的操作符，只是返回值不同。代码清

单3-4在一个泛型方法中对此进行了演示，并给出了“类型推断”和“转换类型约束”的实例。

代码清单3-4　以泛型方式将一个给定的值和默认值进行比较

```
static int CompareToDefault<T>(T value)
   where T : IComparable<T>
{
   return value.CompareTo(default(T));
}
...
Console.WriteLine(CompareToDefault("x"));
Console.WriteLine(CompareToDefault(10));
Console.WriteLine(CompareToDefault(0));
Console.WriteLine(CompareToDefault(-10));
Console.WriteLine(CompareToDefault(DateTime.MinValue));
```

在代码清单3-4中，我们为泛型方法使用了3种不同的类型：`string`、`int`和`DateTime`。`CompareToDefault`方法规定只能使用实现了`IComparable<T>`接口的类型，这样才能为传入的值调用`CompareTo(T)`。传入的值要和类型的默认值进行比较。`string`是引用类型，默认值是`null`——根据有关`CompareTo`的文档，所有引用类型的值都大于`null`，所以第一个结果是1。随后3行与`int`的默认值进行比较，根据结果可以看出`int`的默认值是0。最后一行输出0，可见`DateTime`的默认值就是`DateTime.MinValue`。

当然，如果传递的参数值是`null`，代码清单3-4中的方法就会失败，调用`CompareTo`这行会如期地抛出`NullReferenceException`。不过暂时不要担心，以后会看到，我们还可以使用`IComparer<T>`。

2. 直接比较

虽然代码清单3-4演示了如何进行比较，但我们并非总是愿意限制自己的类型实现`IComparable<T>`或其孪生兄弟接口`IEquatable<T>`，后者提供了一个强类型的`Equals(T)`方法，以弥补所有类型都具备的`Equals(object)`方法的不足。如果没有这些接口允许我们访问的一些额外信息，那么在进行比较时，除了调用`Equals(object)`，就再也没有别的办法了。如果要比较的值是值类型，`Equals(object)`会导致装箱（实际上，在某些情况下，有两个类型可以帮到我们——马上就会讲到它们）。

如果一个类型参数是未约束的（即没有对其应用约束），那么且只能在将该类型的值与`null`进行比较时才能使用`==`和`!=`操作符。不能直接比较两个`T`类型的值。如果类型实参是一个引用类型，会进行正常的引用比较。如果为`T`提供的类型实参是一个非可空值类型，与`null`进行比较的结果总是显示它们不相等（这样一来，JIT编译器就可以移除这个比较）。如果类型实参是可空值类型，那么就会自然而然地与类型的空值进行比较[①]。（如果不明白最后一条有什么意义，先不要担心。当你读到下一章时，就会明白了。有的特性过于“纠缠不清”，以至于我为了讲清楚一个特性，必须引用另一个特性。）

① 在撰写本书的时候（用.NET 4.5及更早的版本测试），JIT编译器生成的比较无约束类型参数值与null的代码对于可空值类型来说特别慢。如果将类型参数T约束为非可空类型，那么比较类型T?的值与null的代码将更快。这指明了未来JIT优化的某些方向。

如果一个类型参数被约束成值类型，就完全不能为它使用==和!=。如果被约束成引用类型，那么具体执行的比较将完全取决于类型参数被约束成什么类型。如果它只是一个引用类型，那么执行的是简单的引用比较。如果它被进一步约束成继承自某个重载了==和!=操作符的特定类型，就会使用重载的操作符。但要注意，假如调用者指定的类型实参恰巧也进行了重载，那么这个重载操作符是不会使用的。代码清单3-5用一个简单的引用类型约束和一个string类型实参对此进行了演示。

代码清单3-5　用==和!=进行引用比较

```
static bool AreReferencesEqual<T>(T first, T second)
   where T : class
{
   return first == second;
}
...
string name = "Jon";
string intro1 = "My name is " + name;
string intro2 = "My name is " + name;
Console.WriteLine(intro1 == intro2);
Console.WriteLine(AreReferencesEqual(intro1, intro2));
```

❶ 比较引用

❷ 使用string重载的==操作符进行比较

虽然string重载了==（❷会输出True），但在❶执行的比较中，是不会使用这个重载的。基本上，编译AreReferencesEqual<T>时，编译器根本不知道有哪些重载可供使用——就好比传入的只是object类型的参数。

并非只有操作符才有这个问题——遇到泛型类型时，编译器会在编译未绑定的泛型类型时就解析好所有方法重载，而不是等到执行时，才去为每个可能的方法调用重新考虑是否存在更具体的重载。例如，Console.WriteLine(default(T));这个语句总是被解析成调用Console.WriteLine(object value)。即使为T传递的类型实参恰好是string，也不会调用Console.WriteLine(string value)。这就好比普通的方法重载是发生在编译时，而不是执行时。但是，那些熟悉C++模板的读者可能会对此感觉有点儿吃惊[①]。

在对值进行比较时，有两个相当有用的类，EqualityComparer<T>和Comparer<T>，两者都位于System.Collections.Generic命名空间中，分别实现了IEqualityComparer<T>和IComparer<T>。这两个类的Default属性能返回一个实现，能为特定的类型采取正确的（比较）操作。

说明　**泛型比较接口**　共有4个主要的泛型接口可用于比较。IComparer<T>和IComparable<T>用于排序（判断某个值是小于、等于还是大于另一个值），而IEqualityComparer<T>和IEquatable<T>通过某种标准来比较两个项的相等性，或查找某个项的散列（通过与相等性概念匹配的方式）。

如果换一种方式来划分这4个接口，IComparaer<T>和IEqualityComparer<T>的实例能够比较两个不同的值，而IComparable<T>和<TIEquatable<T>的实例则可以比较它们本身和其他值。

---

① 在第14章我们可以看到，动态类型提供了在执行时处理重载的能力。

更多的细节请参见文档。执行比较时，请考虑使用这些类型（以及其他相似的类型，比如StringComparer）。我们的下一个例子将使用EqualityComparer<T>。

3. 完整的比较例子：表示一对值

在“实现泛型”（实际上是“实现中级泛型”）这一主题的最后，我将给出一个完整的例子。它实现了一个有用的泛型类型Pair<T1,T2>，用于容纳两个值，类似键/值对，但这两个值之间可以没有任何关系。

> **说明　.NET 4和元组**　.NET 4提供了很多具备这种功能的类型，它们包含不同数量的类型参数。参见System命名空间下的Tuple<T1>、Tuple<T1, T2>等。

除了提供属性来访问值本身之外，我们还覆盖了Equals和GetHashCode方法，从而使这个类型的实例能很好地作为字典中的键来使用。代码清单3-6给出了完整代码。

代码清单3-6　表示一对值的泛型类

```
using System;
using System.Collections.Generic;

public sealed class Pair<T1, T2> : IEquatable<Pair<T1, T2>>
{
   private static readonly IEqualityComparer<T1> FirstComparer =
      EqualityComparer<T1>.Default;
   private static readonly IEqualityComparer<T2> SecondComparer =
      EqualityComparer<T2>.Default;

   private readonly T1 first;
   private readonly T2 second;

   public Pair(T1 first, T2 second)
   {
      this.first = first;
      this.second = second;
   }

   public T1 First { get { return first; } }

   public T2 Second { get { return second; } }

   public bool Equals(Pair<T1, T2> other)
   {
      return other != null &&
         FirstComparer.Equals(this.First, other.First) &&
         SecondComparer.Equals(this.Second, other.Second);
   }

   public override bool Equals(object o)
   {
      return Equals(o as Pair<T1, T2>);
   }
   public override int GetHashCode()
   {
```

```
        return FirstComparer.GetHashCode(first) * 37 +
            SecondComparer.GetHashCode(second);
    }
}
```

代码清单3-6非常直观。成分值[①]（constituent value）被存储到具有恰当类型的成员变量中，并通过简单的只读属性来对它们进行访问。我们实现了`IEquatable<Pair<T1,T2>>`，从而提供一个强类型的API，以避免不必要的执行时检查。相等性和散列码计算都利用了两个类型参数的默认相等性比较器——这样可以自动处理`null`，从而使代码变得简单一些。这里使用静态变量来存储比较`T1`和`T2`的相等比较器，主要是由于篇幅所限而用于代码的格式化，下一节还将用它作为参考。

说明　计算散列码　用于计算散列码的公式参考了Joshua Bloch的*Effective Java*第2版（Addison-Wesley，2008），这个公式基于两个“部分”结果。虽然不能保证这样能获得散列码的良好分布，但就我看来，它比使用按位“异或”要好一些。详情参见Effective Java，书中还提供了其他许多有用的技巧和设计建议。

现在我们有了`Pair`类，该如何构造它的实例呢？这时，你需要使用下面的代码：

```
Pair<int,string> pair = new Pair<int,string>(10, "value");
```

这并不十分理想，要是能使用类型推断就好了，但是那只能用于泛型方法，而且`Pair`类不包含任何泛型方法。如果我们在泛型类型中放入一个泛型方法，那么在调用该方法时仍然需要指定该类型的类型参数。解决方法是使用包含泛型方法的非泛型辅助类，如代码清单3-7所示。

代码清单3-7　使用包含泛型方法的非泛型类型进行类型推断

```
public static class Pair
{
    public static Pair<T1,T2> Of<T1,T2>(T1 first, T2 second)
    {
        return new Pair<T1,T2>(first, second);
    }
}
```

如果你是第一次阅读本书，请忽略静态类型这个声明，等到第7章再介绍。重点是我们拥有了一个包含泛型方法的非泛型类。这意味着可以将之前的示例改写为下面这种优雅的形式：

```
Pair<int,string> pair = Pair.Of(10, "value");
```

在C# 3中，我们甚至可以忽略变量`pair`的显式类型，不过我们还是不要超前了。使用非泛型辅助类（或部分的泛型辅助类，例如有两个以上的类型参数并希望推断其中的一部分而让另一部分保持显式）是一个非常有用的技巧。

到目前为止，“中级”特性已经介绍完了。我知道假如是初次接触，会感觉它过于复杂了一

① 也就是`pair`中的两个值。——译者注

点，但不要轻言放弃：与获得的好处相比，这一点复杂性是微不足道的。熟能生巧。有了Pair类作为参照之后，你或许应该检查一下自己的代码库，核实一下你过去重复实现的设计模式是否只是使用了不同的类型而已。

所有大型的主题都总有更多的知识等着你去学习。下一节将指导你了解泛型中最重要的高级主题。如果感觉现在有点吃力，可以暂时跳到相对比较容易的3.5节。那一节将探讨泛型的一些局限。虽然下一节的主题最终是需要理解的，但假如感觉一切都过于“新”，那么暂时跳到3.5节也无妨。

3.4 高级泛型

你可能希望我在本章剩余的部分，会把迄今为止尚未讲过的有关泛型的一切内容都拿出来讲一讲。但是，涉及泛型时，会有太多琐碎的问题，所以要实现那个目标是不可能的。其实就连我自己都不愿意把所有细节都看一遍，更不用说把它们写到书里了。幸好，微软和ECMA已经将所有细节记录在了语言规范文档中。所以，如果你想了解一些这里没有提到的个别情况，请查阅文档。可惜我无法指出规范的哪一部分涵盖了泛型，因为它们几乎无处不在。如果你的代码恰巧十分复杂，需要参考规范才能明白具体的意图，你就应该将其重构为更明显的方式。你绝不希望所有维护者今生来世都在阅读那些惨不忍睹的细节。

本节的目标是讨论你可能想了解的有关泛型的一切知识。我们会侧重于CLR和框架，不再将重点放在C# 2语言的语法上面，当然这些知识在用C#进行开发时都会很有用。首先讨论泛型类型的静态成员，其中包括类型初始化。接下来，自然是讨论所有这一切在幕后是如何实现的。但是，也不会讲得过于深入，我们只关注这种实现所带来的重要影响。我们将研究一下在C# 2中使用foreach来枚举一个泛型集合时发生的事情。在本节最后，还要讨论泛型对.NET Framework中的“反射”的影响。

3.4.1 静态字段和静态构造函数

就像实例字段从属于一个实例一样，静态字段从属于声明它们的类型。如果在SomeClass中声明了静态字段x，不管创建SomeClass的多少个实例，也不管从SomeClass派生出多少个类型，都只有一个SomeClass.x字段[①]。这在C# 1就很常见，那么它与泛型的关系是怎样的呢？

答案是：每个封闭类型都有它自己的静态字段集。代码清单3-6中也可以看到这一点，我们将默认的T1和T2的相等比较器存储在静态字段里。下面我们通过另一个示例来看看更详细的内容。代码清单3-8创建了一个含有静态字段的泛型类型。我们为不同的封闭类型设置字段的值，然后打印这些值，证明它们是各自独立的。

① 更准确地说，是每个应用程序域（application domain）一个静态字段。考虑到本节的目的，我们假定处理的只有一个应用程序域。处理多个不同的应用程序域时，适用于非泛型类型的概念也适用于泛型。用[ThreadStatic]装饰的变量也违反了这一规则。

代码清单3-8　证明不同的封闭类型具有不同的静态字段

```
class TypeWithField<T>
{
   public static string field;
   public static void PrintField()
   {
      Console.WriteLine(field + ": " + typeof(T).Name);
   }
}
...
TypeWithField<int>.field = "First";
TypeWithField<string>.field = "Second";
TypeWithField<DateTime>.field = "Third";

TypeWithField<int>.PrintField();
TypeWithField<string>.PrintField();
TypeWithField<DateTime>.PrintField();
```

我们将每个字段的值都设为一个不同的值，并打印封闭类型使用的类型实参的名称和每个字段的值。代码清单3-8的输出如下：

```
First: Int32
Second: String
Third: DateTime
```

所以，基本的规则是："每个封闭类型有一个静态字段。"同样的规则也适用于静态初始化程序（static initializer）和静态构造函数（static constructor）。然而，一个泛型类型可能嵌套在另一个泛型类型中，而且一个类型可能有多个泛型参数。虽然听起来很复杂，但它的工作方式与你想象的差不多。代码清单3-9展示了一个例子，这一次是用静态构造函数来演示有多少类型。

代码清单3-9　嵌套泛型类型的静态构造函数

```
public class Outer<T>
{
   public class Inner<U,V>
   {
      static Inner()
      {
         Console.WriteLine("Outer<{0}>.Inner<{1},{2}>",
                           typeof(T).Name,
                           typeof(U).Name,
                           typeof(V).Name);
      }
      public static void DummyMethod() {}
   }
}
...
Outer<int>.Inner<string,DateTime>.DummyMethod();
Outer<string>.Inner<int,int>.DummyMethod();
Outer<object>.Inner<string,object>.DummyMethod();
Outer<string>.Inner<string,object>.DummyMethod();
Outer<object>.Inner<object,string>.DummyMethod();
Outer<string>.Inner<int,int>.DummyMethod();
```

第一次调用DummyMethod()时，不管使用的是什么类型，都会导致Inner类型的初始化，此时静态构造函数打印一些诊断信息。每个不同的类型实参列表都被看做一个不同的封闭类型，所以代码清单3-9的输出如下：

```
Outer<Int32>.Inner<String,DateTime>
Outer<String>.Inner<Int32,Int32>
Outer<Object>.Inner<String,Object>
Outer<String>.Inner<String,Object>
Outer<Object>.Inner<Object,String>
```

和非泛型类型一样，任何封闭类型的静态构造函数只执行一次。所以，代码清单3-9的最后一行不会产生第6行输出。Outer<string>.Inner<int,int>的静态构造函数之前已经执行过了，第2行输出就是它产生的。

为了进一步打消你的疑虑，假如在Outer内有一个非泛型的PlainInner类，那么每个封闭的Outer类型中仍然只有一个Outer<T>.PlainInner类型。所以Outer<int>.PlainInner将独立于Outer<long>.PlainInner，就像前面看到的那样，各自拥有单独的静态字段集。

现在我们知道了一个不同的类型是由什么构成的，接着，应该思考一下这对生成的本地代码数量的影响。并没你想象得那样糟。

3.4.2 JIT编译器如何处理泛型

对于所有不同的封闭类型，JIT的职责就是将泛型类型的IL转换成本地代码，使其能真正运行起来。从某些方面来说，我们并不需要知道具体的转换过程是怎样的——只需留意内存和CPU时间即可。如果JIT为每个封闭类型都单独生成本地代码，就像这些类型相互之间没有任何联系一样，我们将不会感觉出太大差异的。但是，JIT的作者十分聪明，非常有必要看看他们做了什么。

首先看一个简单的、只有一个类型参数的情况。为方便讨论，我们使用List<T>作为例子。JIT为每个以值类型作为类型实参的封闭类型都创建不同的代码。然而，所有使用引用类型（string、Stream、StringBuilder等）作为类型实参的封闭类型都共享相同的本地代码。之所以能这样做，是由于所有引用都具有相同的大小（32位CLR上是4字节，64位CLR上是8字节。但是，在任何一个特定的CLR中，所有引用都具有相同的大小）。无论实际引用的是什么，引用（构成的）数组的大小是不会发生变化的。栈上一个引用所需的空间始终是相同的。无论使用的类型是什么，都可以使用相同的寄存器优化措施，即使是List<Reason>也不例外[①]。

如3.4.1节所述，每个类型还可以有它自己的静态字段，但可执行代码本身是可以重用的。当然，JIT采用的仍然是“懒人”原则——除非需要，否则不会为List<int>生成代码。而一旦生成代码，代码就会缓存起来，以备将来再次使用List<int>。

理论上，至少对一些值类型来说，代码是可以共享的。但JIT必须十分谨慎，不仅要考虑到大小[②]，还要考虑到垃圾回收的问题——JIT必须能快速识别一个struct值中的引用是否是活着

① Reason是一个虚构的类，而所有类都是引用类型。——译者注

② 大小不一致的值类型不能共享代码。——译者注

的[①]。然而，假如值类型具有相同的大小，而且就GC看来具有相同的“内存需求量”，那么是应该能够共享代码的。但到本书完稿时为止，这仍然是一个优先级十分低的需求，所以一直没有实现，而且将来极有可能会一直这样。

虽然一般只有喜欢搞学术研究的人才会对这一级别的细节问题感兴趣，但我要指出的一点是，由于要进行JIT编译的代码增多了，所以确实会对性能造成轻微影响。不过，泛型本身在性能上的优势是相当巨大的，这同样是由于现在有机会将不同的类型通过JIT编译成不同的代码。下面以一个`List<byte>`为例。在.NET 1.1中，为了将单独的字节添加到一个`ArrayList`中，需要对每个字节进行装箱，并存储对每个已装箱值的引用。使用`List<byte>`则无此问题——`List<T>`用一个`T[]`类型的成员数组替代了`ArrayList`中的`object[]`，而且那个数组具有恰当的类型，会占用恰当（大小）的空间。所以，在`List<byte>`中，是直接用一个`byte[]`来存储数组元素。（在许多方面，这使得`List<byte>`在行为上就像是一个`MemoryStream`。）

图3-3展示了一个`ArrayList`和一个`List<byte>`，它们分别包含6个相同的值。数组本身拥有不止6个元素，从而允许扩充。`List<T>`和`ArrayList`都有一个缓冲区，在必要时会创建一个更大的缓冲区。

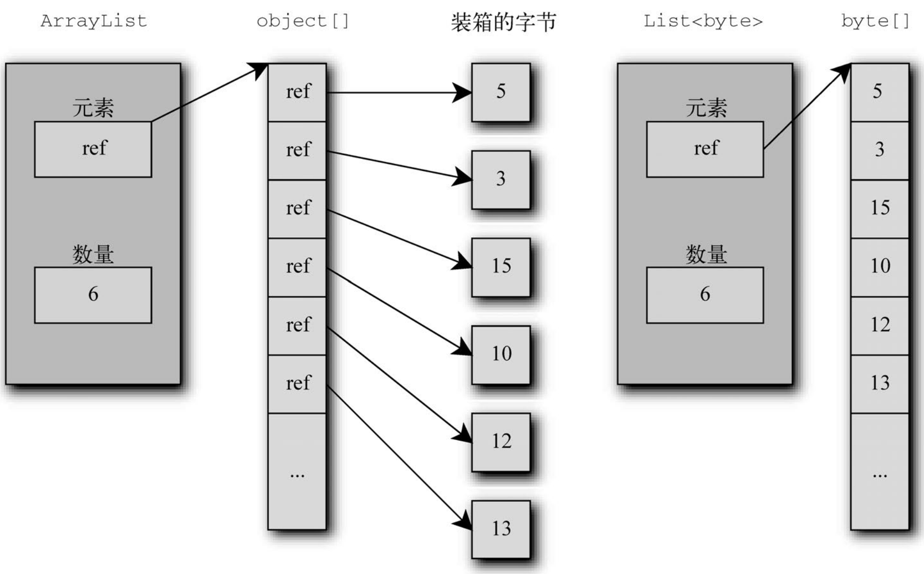


图3-3　演示在存储值类型时，为什么`List<T>`占用的空间比`ArrayList`少得多

两者在效能上的差异令人难以置信。先来看看`ArrayList`，假定使用的是一个32位CLR[②]。每个已装箱的字节都要产生8字节的对象开销，另加4字节（本来是1字节，但要向上取整到一个字的边界）用于数据本身。除此之外，引用本身也要消耗4字节。所以，每个有效数据都要花费

① 如果堆中分配的内存仍被`struct`中的一个字段引用，这一块堆内存就无法进行垃圾回收。——译者注

② 在64位CLR上运行时，开销会更大。

至少16字节。除此之外，缓冲区中还要为引用准备一些额外的未使用的空间。

相比而言，`List<byte>`中的每个字节都占用元素数组中一个字节的空间。缓冲区仍有“浪费”的空间，可用于新增项，但最起码，每个未使用的元素只会浪费一个字节。

我们不仅节省了空间，而且加快了执行速度。现在不需要花时间进行装箱，不需要因为对字节进行拆箱而检查类型，也不需要对不再引用的已装箱值进行垃圾回收。

然而，不需要深入到CLR一级，就能很明显地发现一些正在发生的事情。C#一直致力于通过语法上的快捷方式来简化编程。下一节将研究一个熟悉的、但采用了泛型的例子：用foreach进行遍历。

3.4.3 泛型迭代

对集合执行的最常见的操作之一就是遍历（迭代）它的所有元素。为此，最简单的办法就是使用foreach语句。在C# 1中，为了使用foreach，集合要么必须实现System.Collections.IEnumerable接口，要么必须有一个类似的GetEnumerator()方法，返回的类型含有一个恰当的MoveNext()方法和Current属性。Current属性不一定是object类型，这正是这些额外的规则的全部意义，尽管乍一看比较奇怪。当然，即使在C# 1中，只要有一个自定义的迭代类型，也可以避免在遍历期间装箱和拆箱。

C# 2使过程变得容易了一些。foreach语句的规则得到了扩展，现在还可以使用`System.Collections.Generic.IEnumerable<T>`接口及其搭档`IEnumerator<T>`。它们是旧的枚举接口的泛型等价接口，而且应该优先使用它们，而不是使用非泛型版本。这意味着在遍历由值类型的元素构成的泛型集合（比如`List<int>`）时，根本不会执行任何装箱。如果换用旧接口，虽然在存储列表元素时不会产生装箱开销，但在用foreach取值时，还是要对它们进行装箱。

所有这一切都在幕后进行——你唯一需要的就是以正常的方式使用foreach语句，将集合的类型实参作为迭代变量的类型使用，然后一切都会顺利进行。但是，事情没有就此结束。在少数情况下，当需要为自己的某个类型实现迭代时，你会发现由于`IEnumerable<T>`扩展了旧的IEnumerable接口，所以要实现两个不同的方法[1]：

```
IEnumerator<T> GetEnumerator();
IEnumerator GetEnumerator();
```

能看出问题吗？两个方法只是返回类型不同，而根据C#的重载规则，一般不允许写这样的两个方法。如果你回想起2.2.2节描述了一个类似的情况，那么我们现在可以使用相同的解决方案。如果使用“显式接口实现”来实现IEnumerable，就可以用一个“普通”的方法来实现`IEnumerable<T>`。幸好，由于`IEnumerator<T>`扩展了IEnumerator，所以两个方法可以使用相同的返回值，而且只需调用一下泛型版本，就可实现非泛型方法。当然，实现`IEnumerator<T>`时，你很快就会发现Current属性也存在类似的问题。

---

① 一个基本的原则是，如果没有问题，泛型接口都应该继承对应的非泛型接口，这样可以实现协变性。例如，假如以前为.NET 1.1写的一个函数要获取IEnumerable类型的参数，而现在有了`IEnumerable<T>`。假如`IEnumerable<T>`不继承IEnumerable，这个函数就不能接受`IEnumerable<T>`类型的参数。——译者注

代码清单3-10提供了一个完整的例子，它实现了一个“可枚举”的类，它始终只是从整数0枚举到9。

代码清单3-10　一个完整的泛型枚举——从0枚举到9

```
class CountingEnumerable: IEnumerable<int>
{
   public IEnumerator<int> GetEnumerator()
   {
      return new CountingEnumerator();
   }

   IEnumerator IEnumerable.GetEnumerator()
   {
      return GetEnumerator();
   }
}

class CountingEnumerator : IEnumerator<int>
{
   int current = -1;

   public bool MoveNext()
   {
      current++;
      return current < 10;
   }

   public int Current { get { return current; } }

   object IEnumerator.Current { get { return Current; } }

   public void Reset()
   {
      current = -1;
   }
   public void Dispose() {}
}
...
CountingEnumerable counter = new CountingEnumerable();
foreach (int x in counter)
{
   Console.WriteLine(x);
}
```

❶ 隐式实现**IEnumerable<T>**

❷ 显式实现**IEnumerable**

❸ 隐式实现**IEnumerator<T>.Current**

❹ 显式实现**IEnumerator.Current**

❺ 证明可枚举类型能正常工作

这个例子的输出结果虽然实际用处不大，但很好地展示了为了恰当地实现泛型枚举而必须经受的一些“考验”——至少是在以传统方式来实现时（采用这种方式，如果在一个不恰当的时刻访问Current属性，是不会抛出异常的）。如果你说只是为了打印数字0～9，代码清单3-9的工作量未免太多，我完全同意，而且如果想遍历真正有用的东西，需要的代码量只会更多。幸好，在第6章我们将会看到，C# 2在很多情况下已经极大地简化了需要在枚举器中进行的工作。届时，我会展示一个“完整”版本，你可能会感激为了使IEnumerable<T>扩展IEnumerable而在设计中引入的小革新。然而我并不是说这样设计是错误的。它意味着如果某方法是用C# 1编写的，

接受IEnumerable参数，你也可以将任何IEnumerable<T>类型的变量传递给该方法。现在，这些并没有2005年时那么重要了，但它仍然是一个有用的转换途径。

在代码清单3-10中，我们采用的技巧就是使用了两次显式接口实现。一次是实现IEnumerable.GetEnumerator方法❷，另一次是实现IEnumerator.Current属性❹。两者在实现时，均直接调用了它们的泛型等价成员（分别是❶和❸）。IEnumerator<T>进行的另一项增补是它扩展了IDisposable，所以还必须提供一个Dispose方法。在C# 1的foreach语句中，如果发现枚举器实现了IDisposable，就会为这个枚举器调用Dispose。但在C# 2中不要求在执行时进行检测——如果编译器发现你实现了IEnumerable<T>，就会在循环末尾（一个finally块中）无条件地调用Dispose。许多枚举器实际上都不需要处置（dispose）任何东西，但俗话说“有备无患”，以后总有机会用到它。平常使用枚举器最多的就是foreach语句（❺）；foreach会自动负责Dispose的调用事宜。这常用于在结束迭代时释放资源，例如，对于从文件中读取几行内容的迭代器来说，在调用代码结束循环时，需要关闭文件句柄。

到目前为止，本章关注的实际一直都是编译时的效率问题。接着，我们将目光转到执行时的灵活性上。我们的最后一个高级主题是“反射”。早在.NET 1.0/1.1中，“反射”就是一个比较复杂的主题。引入了泛型类型和泛型方法之后，它变得更复杂了。.NET Framework提供了我们需要的一切（C# 2语言本身也提供了一点儿有帮助的语法）。虽然这种复杂性有点儿让人望而生畏，但假若日积月累，持之以恒，相信最后肯定会有所收获。

3.4.4 反射和泛型

不同的人用反射做不同的事情。可以用它在程序执行时进行对象的“内省”（introspection），从而执行简单的数据绑定。可以用它检查充满各种“程序集”（assembly）的目录，以寻找某个plugin接口的实现。可以为控制反转框架（参见http://mng.bz/xc3J）编写一个文件，来加载和动态配置应用程序的组件。由于反射的应用是如此多样，所以我不会聚焦于其中任何一种特定的应用，而是就如何执行一些常见任务给出一些通用的指导原则。首先来看一下对typeof操作符的扩展。

1. 对泛型类型使用typeof

反射的一切都是围绕“检查对象及其类型”展开的。所以，最重要的就是获取System.Type对象的引用。这样就可以访问与特定类型有关的所有信息。对于编译时已知的类型，C#使用typeof操作符来获取这样的引用。现在，typeof已得到了扩展，可以很好地支持泛型类型。

typeof可通过两种方式作用于泛型类型——一种方式是获取*泛型类型定义*（即“未绑定泛型类型”），另一种方式是获取特定的已构造类型。为了获取泛型类型定义（即没有指定任何类型实参的类型），需要提供声明的类型名称，删除所有类型参数名称，但保留逗号。为了获取已构造类型，需要采取与声明泛型类型变量时相同的方式指定类型实参就可以了。代码清单3-11演示了这两种方式。它使用了一个泛型方法，这样就又可以看到如何为类型参数使用typeof，之前已经在代码清单3-8中见过这样的写法了。

代码清单3-11　对类型参数使用typeof操作符

```
static void DemonstrateTypeof<X>()
{
   Console.WriteLine(typeof(X));                    <-- 显示方法的类型参数

   Console.WriteLine(typeof(List<>));               <-- 显示泛型类型
   Console.WriteLine(typeof(Dictionary<,>));
                                                    ❶ 显式封闭类型（尽管使用了类型参数）
   Console.WriteLine(typeof(List<X>));

   Console.WriteLine(typeof(Dictionary<string,X>));

   Console.WriteLine(typeof(List<long>));           <-- 显式封闭类型
   Console.WriteLine(typeof(Dictionary<long,Guid>));
}
...
DemonstrateTypeof<int>();
```

代码清单3-11的大部分代码你都能读懂，但我仍然要指出两点。首先，请注意获取Dictionary<TKey,TValue>的泛型类型定义的语法。尖括号内的逗号必须保留，它可以告知编译器查找具有两个类型参数的类型。记住，可能同时存在多个同名的泛型类型，它们只是类型参数的数量不同。例如，为了获取MyClass <T1,T2,T3,T4>的泛型类型定义，需要使用typeof(MyClass<,,,>)。在IL中，类型参数的数量是在框架所用的完整类型名称中指定的。在这个完整类型名称的第一部分之后，会添加一个`字符，然后是参数数量。再然后，会在一对方括号(不是我们习惯的尖括号)内显示类型参数。例如，上述代码输出时，第2行会以List`1[T]作为本行输出的结束，表明有一个类型参数。而第3行则为Dictionary`2[TKey, TValue]。

其次，注意任何使用了方法类型参数（X）的地方，执行时都会使用类型实参的实际值。所以，❶会打印List`[System.Int32]，而非你认为的List`1[X][①]。也就是说，编译时还处于开放状态的类型，执行时就可能是封闭的。这很容易使人混淆，假如结果出乎你的预料，就应该意识到这一点。在其他情况下，也不必过于担心。要在执行时获取一个真正开放的已构造类型，还需要更多的工作。请查阅Type.IsGenericType的MSDN文档，其中提供了一个难度适中的例子（http://mng.bz/9w6O）。

下面列出了代码清单3-11的输出，以供参考：

```
System.Int32
System.Collections.Generic.List`1[T]
System.Collections.Generic.Dictionary`2[TKey,TValue]
System.Collections.Generic.List`1[System.Int32]
System.Collections.Generic.Dictionary`2[System.String,System.Int32]
System.Collections.Generic.List`1[System.Int64]
System.Collections.Generic.Dictionary`2[System.Int64,System.Guid]
```

获取了泛型类型的对象之后，接着可以采取许多种操作。以前支持的所有操作（查找类型的成员、创建一个实例，等等）仍然支持，虽然有的操作不适用于泛型类型定义。除此之外，还新增了许多操作，允许你查询类型的泛型特征（generic nature）。

① 我故意没有使用类型参数的常规名称T，因此可以清晰地阐述List<T>声明中的T和该方法中声明的X之间的不同。

2. System.Type的属性和方法

虽然新增许多新的方法和属性，但有两个方法是最重要的：GetGenericTypeDefinition和MakeGenericType。两个方法所执行的操作实际上是相反的——第一个作用于已构造的类型，获取它的泛型类型定义；第二个作用于泛型类型定义，返回一个已构造类型。如果MakeGenericType方法名变为ConstructType、MakeConstructedType或者含有“Construct”或“Constructed”字样的其他名称，它的作用会更加明确。但事已至此，我们只能接受。

和普通类型一样，任何特定的类型只有一个Type对象。所以，如果调用两次MakeGenericType，每次都传递相同的类型作为参数，那么都返回同一个引用。另外，假如用同一个泛型类型定义构造了两个类型，并对这两个类型调用GetGenericTypeDefinition，那么两个调用同样会返回相同的结果。即使构造的类型并不相同（如List<int>和List<string>）。

另一个值得研究的方法是.Net 1.1中就有的Type.GetType(string)以及和它相关的Assembly.GetType(string)方法。这两个方法提供了typeof的一个动态等价物。你可能期望对适当的程序集调用GetType方法，可以得到与代码清单3-11同样的输出结果。可惜，事情并没有这么简单。针对封闭的已构造类型，是可以这样做的——将类型实参放到方括号中即可。但是，对于泛型类型定义，则需要完全删除方括号——否则GetType会认为你指定的是一个数组类型。代码清单3-12演示了所有这些方法的实际应用。

代码清单3-12　获取泛型和已构造Type对象的各种方式

```
string listTypeName = "System.Collections.Generic.List`1";

Type defByName = Type.GetType(listTypeName);

Type closedByName = Type.GetType(listTypeName + "[System.String]");
Type closedByMethod = defByName.MakeGenericType(typeof(string));
Type closedByTypeof = typeof(List<string>);

Console.WriteLine(closedByMethod == closedByName);
Console.WriteLine(closedByName == closedByTypeof);

Type defByTypeof = typeof(List<>);
Type defByMethod = closedByName.GetGenericTypeDefinition();

Console.WriteLine(defByMethod == defByName);
Console.WriteLine(defByName == defByTypeof);
```

代码清单3-12输出4次True，证明无论怎样获取对一个特定类型对象的引用，都只涉及一个这样的对象。

前面提到，Type有许多新的方法和属性，比如GetGenericArguments、IsGenericTypeDefinition和IsGenericType。IsGenericType的文档或许是你展开进一步研究的最佳起点。

3. 反射泛型方法

泛型方法新增了一套类似的（但数量较少的）属性和方法。代码清单3-13进行了简单演示，它通过反射来调用泛型方法。

代码清单3-13　通过反射来获取和调用泛型方法

```
public static void PrintTypeParameter<T>()
```

```
{
   Console.WriteLine(typeof(T));
}
...
Type type = typeof(Snippet);
MethodInfo definition = type.GetMethod("PrintTypeParameter");
MethodInfo constructed = definition.MakeGenericMethod(typeof(string));
constructed.Invoke(null, null);
```

首先获取泛型方法定义，然后使用MakeGenericMethod返回一个已构造的泛型方法。和类型一样，还可以执行其他操作，但和Type.GetType不同的是，没有办法在GetMethod调用中指定一个已构造的方法。另外，对于只是类型参数数量不同的多个重载方法，.NET Framework还存在一个问题：在Type中，没有任何方法允许指定类型参数的数量。所以，只能调用Type.GetMethods（注意多了一个“s”），并在返回的所有方法中查找合适的那一个。

获取了已构造的方法之后，就可以调用了。这个例子中的两个实参都是null，因为我们要调用的是一个静态方法，它不获取任何“普通”的参数。输出结果是System.String，这和我们期望的一样。注意，从泛型类型定义获取的方法不能直接调用——相反，必须从一个已构造的类型获取方法。无论泛型方法还是非泛型方法，这一点都适用。

说明　C# 4可以挽救你　一切看上去凌乱不堪？我同意你的感受。幸好，许多情况下，C#的动态类型可以拯救我们，它在没有泛型反射的情况下进行了大量工作。但动态类型不是万能的，因此你应该注意一下前述代码的一般流程，但是在那些确实需要使用动态类型的地方，一定会收到非常出色的效果。第14章会详细介绍动态类型。

同样，MethodInfo也新增了一些方法和属性。从MSDN中查阅的话，IsGenericMethod是一个很好的起点（参见http://mng.bz/P36u）。希望本节提供的信息能够足够让你上手。另外，本节还指出了你没预料到的一些额外的复杂性，但在开始学习利用反射来访问泛型类型和方法时，就会遇到。

好了，关于“高级特性”，就讲这么多。重申一下，本书不打算写成一本面面俱到的参考指南，但事实上，大多数开发者也不需要做到“面面俱到”。对于你来说，我希望你是这些开发者中的一员，因为（语言）规范读得越深入，就越难理解。记住，除非你独自工作，只为自己工作，否则代码极有可能被多人使用。如果你需要在代码中使用比这里演示的还要复杂的特性，那么你不应该假设别人在没有帮助的情况下，就可以理解你的代码。另一方面，如果发现你的同事不知道我们迄今为止讨论的一些主题，请随时带他们去最近的书店……

本章最后一节将讨论C#泛型的一些限制，以及其他语言中的类似特性。

3.5　泛型在 C#和其他语言中的限制

毋庸置疑，泛型在表现力（expressiveness）、类型安全性以及性能方面对C#贡献良多。这个特性进行了精心设计，可以处理C++程序员平时用模板来完成的大多数任务，同时避免了模板的

一些缺点。但是，这并不是说它就不存在什么限制了。有的问题C++模板能轻松解决，但C#泛型无能为力。同样，虽然Java的泛型通常要弱于C#的泛型，但有一些概念能在Java中表示，但在C#中却不行。本节将指导你认识一些最常见的缺点，同时将C#/.NET所实现的泛型同C++模板和Java泛型进行简单比较。

要强调的一点是，虽然我会指出这样或那样的问题，但这并不表示这些问题一开始就应该避免。换成我来做，并不见得能做得更好！语言和平台的设计者必须在功能与复杂性之间取得平衡（而且还要尽快完成设计和实现，时间不能拖得太久）。极有可能的是，你根本不会遇到这些问题。即使遇到了这些问题，也可以凭借这里提供的指导原则来解决它们。

首先回答一个你迟早都会提出的问题：为什么不能将`List<string>`转换成`List<object>`？

3.5.1 泛型可变性的缺乏

2.2.2节探讨了数组的*协变性*——引用类型的数组可以被视为它的基类型的数组，或者被视为它所实现的任何接口的数组。实际上这一概念有两种形式，称为*协变性*和*逆变性*，或统称为*可变性*。泛型不支持可变性——它们是*不变体*（invariant）。这是为类型安全性着想，但它有时也会带来不便。

需要明确指出的是：泛型可变性在C# 4中有了一定程度的改善。但这里列出的诸多限制仍然有效，而且你可以将本节看成是对可变性概念有用的介绍。第13章将介绍C# 4如何改善了可变性，不过很多泛型可变性的示例都依赖于C# 3中的其他新特性，包括LINQ。可变性本身就是一个相当复杂的主题，因此你应该在弄懂C# 2和C# 3的其他内容之后再来学习它。为了增加可读性，我不会在本节指出它们与C# 4稍微不同的所有地方。一切都将在第13章揭晓。

1. 泛型为何不支持协变性

假定有两个类，分别是`Turtle`（海龟）和`Cat`（猫），两者都派生于`Animal`。在下面的代码中，数组代码（左侧）是有效的C# 2代码，而泛型代码（右侧）不是：

有效（在编译时）	无　　效
`Animal[] animals = new Cat[5];` `animals[0] = new Turtle();`	`List<Animal> animals = new List<Cat>();` `animals.Add(new Turtle());`

在两种情况下，编译器对于第2行代码都是没有任何问题的。但是，右侧的第一行会造成以下错误：

```
error CS0029: Cannot implicitly convert type
      'System.Collections.Generic.List<Cat>' to
      'System.Collections.Generic.List<Animal>'
```

这是框架和语言的设计者在仔细考虑后做出的选择。你肯定要问为什么要禁止——答案在于第2行。第2行表面看没有任何问题。毕竟，`List<Animal>`确有一个方法的签名是`void Add(Animal value)`——例如，一个`Turtle`当然能放到任何动物列表中。然而，在左侧的代码中，`animals`实际引用的对象是一个`Cat[]`；在右侧的代码中，`animals`实际引用的是一个`List<Cat>`。两者都只要求存储对`Cat`实例的引用。左侧的数组版本虽然可以通过编译，但执行

时一样会失败。泛型的设计者认为，这比编译时就失败还要糟糕——静态类型的全部意义就在于在代码运行之前找出错误。

说明 **那么，为什么数组是协变的？** 在回答了为什么泛型是不变的问题之后，**接下来**的一个问题自然就是为什么数组是协变的？根据*Common Language Infrastructure Annotated Standard*（Addison-Wesley Professional，2003年版），在语言的第一个版本中，设计者希望面向尽可能多的用户，为此采取的策略之一就是就是允许支持从Java中编译来的代码。换句话说，.NET之所以有协变数组，是因为Java有协变数组——虽然这个功能在Java中是一个公认的"瑕疵"。

这就是这些问题的原因。但是，为何要关心这个？怎样才能解决存在的限制呢？

2. 协变性在什么时候有用

前面列表的例子显然是有问题的。我们可以向列表中添加项，但也因此失去了类型安全。添加操作是向API输入值，值由调用者提供。如果我们人为限制只能输出值，将会发生什么呢？

最明显的例子就是`IEnumerator<T>`和（相关的）`IEnumerable<T>`。实际上，它们是泛型协变最典型的示例。它们共同描述一个值的序列，每一个值都与`T`兼容，因此可以写成这样

```
T currentValue = iterator.Current;
```

这里只使用了普通的兼容概念，例如，一个`IEnumerator<Animal>`可以产生对`Cat`或`Turtle`实例的引用。我们无法添加与实际序列类型不符的值，因此希望能够将`IEnumerator<Cat>`看成是`IEnumerator<Animal>`。举一个例子看看可以在什么地方使用这种想法。

假设使用自定义的形状示例来描述继承关系，其中包含一个接口（`IShape`）。现在考虑另一个由形状组成的图形接口`IDrawing`。我们拥有两个具体的图形类型——`MondrianDrawing`（由矩形组成）和`SeuratDrawing`（由圆形组成）[①]。图3-4展示了相关类的继承关系。

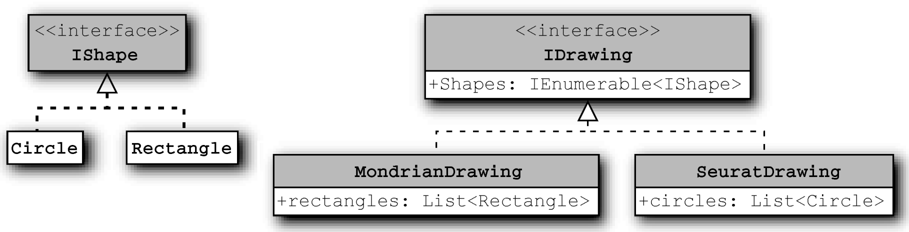


图3-4　形状和图形的接口，以及各自的两个实现

---

① 如果你对这些名称一头雾水（指Mondrian和Seurat，这两个人都是画家。——译者注），可以参阅维基百科中的条目（http://mng.bz/pWl7和http://mng.bz/1025）。它们对我来说还有着特殊的意义：Mondrian是我在Google使用的代码复查工具，而Seurat也是Stephen Sondheim的经典音乐剧《星期天与乔治同游公园》（*Sunday in the Park with George*）中乔治的姓。

两个图形类型都需要实现IDrawing接口，所以需要公开具有以下签名的属性：

```
IEnumerable<IShape> Shapes { get; }
```

然而如果每个图形类型都在内部维护一个更强类型的列表，将会更加简单。例如，Seurat图形可以包含一个List<Circle>类型的字段，这对它来说比List<IShape>要有用得多，因为如果需要以圆特有的方式来操作圆形时，就不必进行强制转换了。如果是List<IShape>，我们可以直接返回或将其包装在ReadOnlyCollection<IShape>中，以防止调用者通过强制转换破坏其状态——这两种实现都代价低廉且十分简单。但如果类型不匹配，就不能这么做了。我们无法将IEnumerable<Circle>转换为IEnumerable<IShape>。那我们能做些什么呢？

以下是一些可以进行的选择。

- 将字段类型改为List<IShape>并保留强制转换。但这并不优雅，而且丧失了泛型的很多优势。
- 使用C# 2提供的实现迭代器的新特性，第6章将详细介绍。这是一个合理的解决方案，但仅限于处理IEnumerable<T>这种情况。
- 在Shapes属性的实现中创建列表的新副本，简便起见可以使用List<T>.ConvertAll。尽管在API中，为集合创建一个独立的副本通常没有任何问题，但在很多情况下，大量的复制会导致不必要的效率降低。
- 将IDrawing改为泛型接口，指明图形中的形状类型。因此ModrianDrawing将实现IDrawing<Rectangle>、SeuratDrawing将实现IDrawing<Circle>。该方法只有在你拥有这个接口时才能使用。
- 创建一个辅助类将IEnumerable<T>适配成另一个：

```
class EnumerableWrapper<TOriginal, TWrapper> : IEnumerable<TWrapper>
   where TOriginal : TWrapper
```

同样，由于这种情形（IEnumerable<T>）的特殊性，我们可以使用一个工具方法。事实上，.NET 3.5发布了两个有用的方法：Enumerable.Cast<T>和Enumerable.OfType<T>。它们属于LINQ，第11章将介绍。尽管这是一种特殊情况，但却可能是最常遇到的泛型协变性的形式。

遇到协变性问题时，可能需要考虑上述所有对策，以及你能想到的其他对策。基本上，你需要具体情况具体分析。可惜，协变性并不是我们必须考虑的唯一问题。还有*逆变性*的问题，你可以把它想象成反方向的协变性。

3. 逆变性在什么地方有用

在感觉上，逆变性不像协变性那样直观，但它确实是合情合理的。使用协变性，可以将SomeType<Circle>转换为SomeType<IShape>（上面那个例子中的SomeType为IEnumerable<T>）。而逆变性则是进行反向转换——从SomeType<IShape>转换为SomeType<Circle>。这怎么可能安全呢？实际上，当SomeType只描述返回类型参数的操作时，协变就是安全的；而当SomeType只描述*接受*类型参数的操作时，逆变就是安全的[①]。

① 这里是协变和逆变的一般原则，第13章有更详细的描述。

只将类型参数放在输入位置的最简单类型就是IComparer<T>，它常用于集合排序。我们来扩展IShape接口（之前没有任何成员），使其包含Area属性。那么很容易写出IComparer<IShape>的一个实现，它能按面积进行排序。然后，你可能会写出下面这样的代码：

```
IComparer<IShape> areaComparer = new AreaComparer();
List<Circle> circles = new List<Circle>();
circles.Add(new Circle(Point.Empty, 20));
circles.Add(new Circle(Point.Empty, 10));
circles.Sort(areaComparer);
```

但这些代码无法工作，因为List<Circle>的Sort方法实际获取的是一个IComparer<Circle>。虽然我们的AreaComparer能比较任意形状，而非只是圆，但编译器对此无动于衷。它会认为IComparer<Circle>和IComparer<IShape>是完全不同的类型。这真让人抓狂，不是吗？如果Sort方法的签名变成下面这样就好了：

```
void Sort<T>(IComparer<S> comparer) where T : S
```

很遗憾，这不仅不是Sort的签名，也不能成为Sort的签名——约束是无效的，因为它是对T而不是S的约束。我们希望能有一个转换类型约束，但要求是在相反的方向，将S约束成T的继承树上方而不是下方的某个位置。

既然这是不可能的，那么我们能做什么？这一次，可供选择的对策要少一些。首先，我们可以重新考虑创建一个泛型辅助类，参见代码清单3-14。

代码清单3-14　使用泛型辅助类解决逆变性缺乏问题

```
class ComparisonHelper<TBase, TDerived> : IComparer<TDerived>
    where TDerived : TBase
{
    private readonly IComparer<TBase> comparer;

    public ComparisonHelper(IComparer<TBase> comparer)
    {
        this.comparer = comparer;
    }

    public int Compare(TDerived x, TDerived y)
    {
        return comparer.Compare(x, y);
    }
}
```

❶ 恰当地约束类型参数

❷ 保存原始的比较器

❸ 使用隐式类型转换来调用比较器

同样，这也是适配器模式在起作用，虽然没有将一个接口写成完全不同的接口，但将IComparer<TBase>改写成IComparer<TDerived>。我们只是存储了原始的比较器，它提供了比较基类型的真正逻辑❷，然后在比较派生类型的项时调用该比较器❸。不需要任何强制转换（即使对隐藏的比较器来说也是如此），这给了我们信心：这个辅助类完全是类型安全的。可以调用基比较器，因为在类型约束中指定了TDerived继承自TBase❶，可以将前者隐式转换为后者。

第二个对策是使面积比较类成为一个泛型类，并添加一个派生约束，使它能比较相同类型的任意两个值，只要该类型实现了IShape即可。对于那些不需要这种功能的情况来说，简便起见，可以保留非泛型类，让它继承这个泛型类：

```
class AreaComparer<T> : IComparer<T> where T : IShape

class AreaComparer : AreaComparer<IShape>
```

当然，只有在能够更改比较类时，才可以这样做，但只要能更改比较类，这就是一个很好的解决方案，但还是感觉不太自然——行为并没有什么不同，但我们为什么要以不同的方式为不同的类型构建比较器呢？我们并没有对行为进行任何特殊化，但为什么要为了简化而从泛型类派生呢？

注意，在为了实现协变性和逆变性而采取的各种方案中，我们使用了更多的泛型和约束，以一种更泛化的方式来表示接口，或者提供泛型的“辅助”类。我知道，在添加了一个约束之后，会使它看起来不再那么泛化。但是，由于我们先使类型或方法成为泛型的，所以已经增加了“泛化性”或者“一般性”（generality）。遇到这样的问题时，应考虑的第一个选择就是通过一个恰当的约束来添加一定程度的“泛化性”。这时，泛型方法（而非泛型类型）一般都可以提供帮助，因为类型推断会让人觉察不到缺乏了“变化”。在C# 3中尤其是这样的，它提供了比C# 2更强的类型推断能力。

这个限制[①]是C#讨论组“热议”的一个话题。其他问题则“学术性”较强，或者只影响到少数开发群体。下一个问题主要影响那些需要在工作中做大量计算的人（一般是科学或金融计算）。

3.5.2 缺乏操作符约束或者“数值”约束

在涉及繁重数学计算时，C#并不是没有弊端。除了最简单的算术运算，其他所有运算都要显式地使用`Math`类。另外，C#还缺乏C风格的`typedef`（使用`typedef`定义了一个数据表示之后，就可以在一个程序的各个地方使用它，并可以轻松地更改）。所有这些都妨碍了C#在科学计算群体中的普及。泛型无法彻底解决上述两个弊端，并且由于一个普遍存在的问题，造成泛型不能充分发挥其作用。

来看看下列（非法的）泛型代码：

```
public T FindMean<T>(IEnumerable<T> data)
{
   T sum = default(T);
   int count = 0;
   foreach (T datum in data)
   {
      sum += datum;
      count++;
   }
   return sum / count;
}
```

上述代码显然并不适用于所有类型的数据——例如，两个`Exception`相加是什么意思呢？显然，这里需要某种形式的约束，它要能表示出我们希望做的事情：将`T`的两个实例加到一起，而且要用一个整数来除`T`。如果准许这样做，那么即使只是限于内建的类型，也能写出一个泛型算

① 指“协变性和逆变性的缺乏”。——译者注

法，让这个算法不必关心操作的是int、long、double，还是decimail。

虽然限制成内建类型有点儿令人失望，但有总比没有好。理想的解决方案是允许用户自定义的类型也以数值的方式进行处理——例如，可以定义一个Complex类型来处理复数[①]。这样一来，复数就可以通过一种泛型方式来存储它的各个“部”，如Complex<float>、Complex<double>等。

有两个（虚构的）解决方案是相互关联的。一个是简单地允许为操作符添加“约束”，这样就可以写出像下面这样的一系列约束（目前是无效的）：

```
where T : T operator+ (T, T), T operator/ (T, int)
```

这就规定了T必须支持早先的代码所要求的操作。另一个解决方案是定义几个操作符，另外可能还要定义一些转换。一个类型为了满足额外的约束，必须支持这些操作符和转换。可以把它称为“数值约束”，写成：where T : numeric。

但是，这两个解决方案都存在一个问题：它们不能表示成普通的接口。这是由于操作符重载是用静态成员来执行的，而静态成员不能用来实现接口。静态接口的概念似乎很吸引人：它们只能声明静态成员，包括方法、操作符和构造函数。这种静态接口可能只在类型约束中有用，但却能允许你以类型安全的泛型方式来访问静态成员。不过这只是一个天马行空般的想法（参见http://mng.bz/3Rk3），不知道未来的C#版本是否会包含这一特性。

有两个迄今为止最巧妙的解决方法，它们需要新版的.NET：一个由Marc Gravell设计（http://mng.bz/9m8i），使用表达式树（将在第9章介绍）来构建动态方法；另一个使用C# 4的动态特性。第14章将给出关于后者的一个示例。不过，正如我所描述的，它们都是动态的——只有等到执行时才能看到你的代码是否能够处理特定的类型。还有一些仍然使用静态类型的解决方案，但它们也有其他缺点（令人不解的是，它们有时竟然比动态代码还慢）。

前面讨论的两个限制是相当实际的——在实际开发期间完全可能遇到这些问题。然而，假如你和我一样有强烈的求知欲，那么可能想知道是否还存在其他限制。这些限制不一定会影响开发速度，但我们仍然感到好奇。特别是，为什么泛型只限于类型和方法？

3.5.3　缺乏泛型属性、索引器和其他成员类型

前面已讨论了泛型类型（包括类、结构、委托和接口）和泛型方法。还有其他许多成员可以参数化。但是，不存在泛型的属性、索引器、操作符、构造函数、终结器[②]或事件。

让我解释一下这句话的意思：很明显，索引器的返回类型可以是一个类型参数——List<T>就是一个典型的例子。属性也是如此，如KeyValuePair<TKey,TValue>。但是，对于一个索引器或属性（或者上述列表中的其他任何成员）来说，它们不可以有额外的类型参数。

① 这里假设你使用的不是.NET 4，因为.NET 4中已经有了System.Numerics.ComplexNumber类型。

② 在C#中，编译器自动将析构函数转换成对Object.Finalize方法的一个覆盖。编译器生成的Finalize方法将析构函数的主体包含到一个try块中，后跟一个finally块来调用基类的Finalize方法。这样就确保一个析构函数总是调用它的基类析构函数。需要注意的是，只有编译器才能进行这个转换。你不能亲自覆盖Finalize，也不能亲自调用Finalize。——译者注

暂时不去管可能有哪些声明语法，先来看看这些成员可能的调用方式：

```
SomeClass<string> instance = new SomeClass<string><Guid>("x");
int x = instance.SomeProperty<int>;
byte y = instance.SomeIndexer<byte>["key"];
instance.Click<byte> += ByteHandler;
instance = instance +<int> instance;
```


相信你和我一样，都认为所有这些语法都有点儿“傻”。终结器甚至不能在C#代码中显式地调用，所以在上述代码中没有专门为它列出一行。就我看来，虽然上面列出的任何一件事情都不能做，但在实际应用中，它们造成的问题并不大。如上一节最后所说的，只是出于学术研究的目的，我们才对这些限制有点儿兴趣。

这个限制最讨厌的地方可能就是不能使用泛型的构造函数。然而，只需在类中使用一个静态的泛型方法，就可以简单地解决这个问题。不过，这就要求使用两套类型实参列表，光想想就觉得十分可怕。

虽然这些并不是C#泛型唯一的限制，但我相信它们是你最有可能遇到的，要么会在日常工作中、在社区讨论中遇到，要么将此特性视为一个整体时遇到。在接下来的两个小节中，你会看到其中的一些限制在另外两种语言中根本就不是问题。我们经常拿C#的泛型和这两种语言的对应特性进行对比：C++（和它的模板进行对比）以及Java（和它的泛型进行对比：Java的泛型是Java 5开始引入的）。首先来看看C++。

3.5.4 同C++模板的对比

C++模板有点儿像是发展到极致的宏。它们非常强大，但代价就是代码膨胀和不容易理解。

在C++中使用一个模板时，会为那一套特定的模板实参编译代码，好像模板实参本来就在源代码中一样。这意味着对约束的需求就不像对编译器需求那么多，因为编译器在为这一套特定的模板实参编译代码时，会检查你可以对类型执行哪些处理。不过，C++标准协会已经意识到约束仍然是有用的。约束又被包含了进来，然后在C++11（C++的最新版本）中删除了，也许未来某一天会重见天日，并采用概念这个名称。

C++编译器具有一定的“智能”，针对任何给定的一套模板实参，代码都只会编译一次。但是，它还没有“聪明”到能共享代码的程度，就像CLR对引用类型所做的那样。不过，不共享代码也是有好处的——它允许进行类型特有的优化，比如为一部分类型参数内联方法调用。但是，针对来自同一个模板的另一部分类型参数，则不内联方法调用。它还意味着能单独为每一套类型参数执行重载决策。而在C#的情况下，只能根据由现有约束为C#编译器提供的有限信息进行一次重载决策。

不要忘记，对于“普通”C++来说，编译是一次完成的，而不是像.NET模型那样，先“编译成IL”，再由“JIT编译成本地代码”。一个C++程序以10种不同的方式使用一个标准模板，就会在程序中包含代码10次。但是，在C#中，一个类似的程序如果以10种不同的方式使用来自框架的一个泛型类型，那么根本不会包含泛型类型的代码。相反，它只是引用一下泛型类型。执行时，需要多少个不同的版本，JIT就会编译多少个，就像3.4.2节描述的那样。

C++模板做得比C#泛型好的一个地方在于，模板实参不要求必须是类型名，变量名、函数名和常量表达式也是允许的。一个常见的例子是使用一个缓冲区类型，它将缓冲区的大小作为模板实参之一。所以，buffer<int,20>是包含20个int值的缓冲区，而buffer<double,35>是包含35个double值的缓冲区。这个功能对于模板元编程（metaprogramming，参见http://en.wikipedia.org/wiki/Template_metaprogramming）是至关重要的。作为一种高级的C++技术，模板元编程的基本思路令我感到“害怕”。但放在专家的手里，它又确实能发挥出强大的作用。

C++模板在其他方面也更加灵活。它们没有3.5.2节描述的问题。另外，其他几个限制在C++中也是不存在的：可以从一个类型参数派生出一个类，而且可以为一套特定的类型实参专门使用一个模板。利用后一种能力，模板作者可以写一些常规代码。假如没有更多的信息，就使用这些常规代码。但是，特定的类型就使用特定的（而且一般是高度优化过的）代码。

.NET泛型存在的可变性问题也存在于C++模板中。由于同样的原因在Vector<shape*>和Vector<circle*>之间不存在隐式转换，但Bjarne Stroustrup给出了一个例子，通过类似的推理，可以将格格不入的东西放在一起。

要想更多地了解C++模板，我推荐Stroustrup的*The C++ Programming Language*第3版（Addison-Wesley Professional，1997年版）。虽然这并不是一本很容易懂的书，但关于模板的那一章讲得相当透彻（当然，你要先熟悉C++的术语和语法）。要了解它与.NET泛型的更多对比，请阅读Visual C++团队关于这个主题的博客文章：http://mng.bz/En13。

在泛型方面，有必要和C#进行比较的另一种语言就是Java。这个特性是从Java 1.5[1]开始引入主流（Java）语言的。但在此之前的几年时间里，已经有好几个项目创建了支持泛型的Java风格的语言。

3.5.5　和Java泛型的对比

C++为模板生成的代码要比C#为泛型生成的代码多一些。而Java生成的则要少一些。事实上，Java运行时根本不知道有泛型的存在。在为泛型类型生成的Java字节码（bytecode，大致和IL等价的一个术语）中，包含一些额外的元数据，来表示这是一个泛型。但是，在编译之后，负责调用的代码根本就发现不了曾有泛型出没的迹象。另外有一点可以肯定，泛型类型的实例只知道它自己非泛型方面的情况。例如，HashSet<E>的实例并不知道它自己被创建成一个HashSet<String>还是一个HashSet<Object>。编译器只是在必要的地方添加强制类型转换，并执行更稳妥的检查。

下面是一个例子——首先是泛型Java代码：

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("hello");
String entry = strings.get(0);
strings.add(new Object());
```

然后是等价的非泛型代码：

① 或者5.0，取决于你使用的是什么编号系统。不要逼我开始这个话题。

```
ArrayList strings = new ArrayList();
strings.add("hello");
String entry = (String) strings.get(0);
strings.add(new Object());
```

两者将生成相同的Java字节码，但最后一行除外——它在非泛型的例子中是有效的，但在泛型版本中会被当做一个错误由编译器“捕捉”。可以将泛型类型作为一个“原始”（raw）类型使用，它相当于为每个类型实参都使用`java.lang.Object`。这种重写（会丢失信息）被称为类型擦除（type erasure）。Java没有用户自定义的值类型，但就连内建的值类型[①]也不能作为类型实参使用。相反，必须使用“已装箱”的版本，例如，对一个整数列表来说，就是`ArrayList<Integer>`。

如果你认为这和C#的泛型相比要“逊色”一些，肯定没人怪你，但Java泛型也提供了一些出色的特性。

- 运行时虚拟机不知道关于泛型的一切，所以只要没有使用旧版本中不存在的类或方法，那么即使在代码中使用了泛型，编译之后代码一样可以在旧版本上运行。.NET的版本控制总体来说要严格得多——对于你引用的每个程序集，都可以指定版本号是否必须精确匹配。除此之外，生成时指定要在2.0 CLR上运行的代码不能在.NET 1上运行。
- 不需要学习一套新的类就可以使用Java泛型。非泛型开发者仍然使用`ArrayList`，泛型开发员只需使用`ArrayList<E>`。现有的类可以轻松地“升级”到泛型版本。
- 以前的特性通过反射系统被有效地利用——`java.lang.Class`（`System.Type`的等价物）是泛型，它允许对编译时类型安全性进行扩展，以覆盖涉及反射的许多情形。然而，在其他一些情况下，它也会带来不便。
- Java使用通配符来支持协变性和逆变性。例如，`ArrayList<? extends Base>`可以理解成：“这是从`Base`派生的某个类型的`ArrayList`，但我们不知道确切是什么类型。”第13章讨论C# 4的泛型可变性时，还会运用一个简短的示例重温一下这一点。

我个人的观点是，.NET泛型几乎在所有方面都要“略胜一筹”。只有遇到协变性/逆变性问题时，我才会想到要是有通配符该有多好！C# 4受限的泛型可变性从某种程度上改善了这一点，但Java的可变性模型在某些时候仍然要更好一些。虽然有泛型的Java比没有泛型的Java好得多，但Java泛型没有性能上的优势，而且安全性只能在编译时予以保证。

3.6 小结

泛型在实际使用时要比描述得简单一些，这是一件好事。虽然泛型可能变得很复杂，但人们普遍认为它是C# 2最重要的一项增补，而且作用相当大。会用泛型写代码之后，再回过头去使用C# 1，你会发现自己其实已经无可救药地爱上了泛型。（幸好，这样的可能性越来越小。）

本章本来就没有打算对泛型的使用进行面面俱到的讲述——那是语言规范的职责，而且如果真的把那些内容都写上，读起来会枯燥无味。相反，我采用了一种偏实用的写作方式，提供了你平时使用泛型时需要的信息，并讨论了一些与学术研究有关的、浅显的理论。

① 指`int`、`long`这样的Java基本类型。——译者注

我们已经知道了泛型的3个主要优点：编译时的类型安全性、性能和代码的表现力。IDE和编译器能提前验证你的代码，这无疑是一件好事情，但也有一些人认为，如果工具能提供智能的选项，能根据实际涉及的类型来进行安全保障，那么应该会更好一些。

值类型在性能上的获益是最大的。在强类型的泛型API（尤其是.NET 2.0提供的泛型集合类型）中使用时，它们不再需要装箱和拆箱。引用类型的性能也有所提升，只是幅度较小。

使用泛型，代码能更清楚地表达其意图，而不是只能通过一条注释或者一个很长的变量名来准确地描述涉及的类型。现在，类型本身的细节就可以表达你的意图了。随着时间的推移，使用注释和变量名的方法经常会变得不够准确。这是因为当代码发生改变后，注释和变量名可能没有相应地进行修改，所以它们对类型的描述可能是不正确的。

当然，泛型并不是万能的，本章描述了它们的一些限制。不过，如果你真的喜欢上了C# 2，并体会到了.NET 2.0 Framework中的泛型类型的妙处，就必然会在自己的代码中频繁地使用它们。

后面的章节还会经常提到泛型，因为有一些新特性是基于它来构建的。事实上，下一章的主题就和泛型密不可分——我们将探讨由`Nullable<T>`实现的可空类型。

第 4 章

可空类型

4

本章内容

- 空值存在的原因
- 可空值的框架和运行时支持
- C# 2的语言支持
- 使用可空类型的模式

多年来，人们对"可空性"（nullity）这一概念的争论时有发生。空引用是表示一个值，还是表示没有值？没有值算是一种有效的值吗？语言是否应该支持可空性这一概念，或者应该用其他方式来表示？

本章会尽量从实用的角度进行讨论，而不打算把它变成一次哲学辩论。首先看看这个问题最初是如何产生的——为什么在C# 1中不能将一个值类型的变量设为`null`，以及一般有哪些备选方案。然后，`System.Nullable<T>`将闪亮登场。在这之后，我们要讲一讲C# 2如何简化可空类型的使用。类似于泛型，可空类型有一些用法出乎你的意料。本章末尾将演示这样的几个例子。

那么，一个值在什么时候不是一个值呢？下面我们就来揭开谜底。

4.1 没有值时怎么办

C#和.NET的设计者不会仅因为喜好就添加一个特性。必须是真正的、重大的问题需要修正，他们才会修改C#语言或者.NET平台。而我们要解决的这个问题，经常可以在C#和.NET讨论组中看到，它可以归纳为：

我想把我的`DateTime`[①]变量设为`null`，但编译器不允许，该怎么办？

这是再自然不过的一个问题。在电子商务应用程序中，用户查看其账户历史时，就极有可能遇到这个问题。如果一份订单虽然已经下单，但尚未发货，那么虽然有一个购买日期，但没有发货日期——在用于描述订单细节的类型中，如何对此进行表示呢？

① 每次问到的几乎都是`DateTime`类型，而不是其他值类型。我也不能完全确定这是为什么——开发者似乎天生就明白为什么一个字节不应该为空，但在遇到日期时，自然而然地会觉得它"本来就允许为空"。

对于C# 2以前的版本，这个问题通常是分成两部分来回答的：首先，为什么不能一开始就使用null呢？其次，有哪些备选的方案？现如今，答案通常都是使用可空类型。但看看C# 1中的解决方案，对于理解问题产生的原因还是非常有价值的。

4.1.1　为什么值类型的变量不能是**null**

如第2章所述，对于一个引用类型的变量来说，其值是一个引用。而值类型变量的值是它本身的真实数据。可以认为，一个非空引用值提供了访问一个对象的途径。然而，null相当于一个特殊的值，它意味着*我不引用任何对象*。

如果把引用想象为URL，null就大致相当于about:blank。内存中用全零来表示null（这也解释了为什么所有引用类型的默认值是null——清除一整块内存的开销最低，所以对象选择用这种方式来初始化），但它本质上采用的是和其他引用一样的方式来存储的。引用类型的变量没有在任何地方隐藏一个“额外的bit”。这意味着不能将“全零”的值用于一个“真正”的引用，但这不是问题——在有那么多的活动对象之前，我们的内存早就用光了[1]。这就说明了为什么null不是有效的值类型的值。

为便于讨论，假定有一个byte类型。byte变量的值用单独一个字节来存储——为了对齐，可能会进行一些填充。但是，值本身在概念上只由一个字节构成。我们可以将值0～255存储到那个变量中，如果试图将超出这个范围的值存储到其中，那么读取到的就是“垃圾”数据。所以，256个“普通”值加1个null值，我们总共要处理257个值。没有办法用一个字节存储那么多的值。为此，设计者可能决定，为每个值类型都设置一个额外的标志位（flag bit），这个标志位用于判断一个值是null还是一个“真正”的值。但这样一来，内存的消耗将急剧增加，更不用说每次想要使用值时，都得对这个标志位进行检查。所以，简单地说，对于值类型，我们希望它拥有一套和真正的值一样完整的位模式（bit pattern），而对于引用类型，则希望丢失一个潜在的值，换取使用一个null值的好处。

这是最普通的情况——那么，为什么有时还是想把一个值类型表示成null呢？最常见的原因是数据库一般会允许为每个类型使用NULL值（除非你有意指定一个字段不允许为空）。因此，你可以在数据库中使用可空的字符数据、可空的整数、可空的布尔值——所有类型都可空。从数据库获取数据时，通常不能容忍有信息丢失的情况发生。所以，就希望有一种方式能表示自己读取的东西是“空”。

但是，这带来了更多的疑问。为什么数据库就允许为日期、整数等使用空值呢？空值通常用于表示未知或缺失的值，比如在前面的电子商务的例子中提到的发货日期。“可空性”代表明确定义的信息“不存在”，而这种信息有许多时候都可能是相当重要的。确实，可空类型并不是只有在处理数据库时才有用：这只是开发者遇到类似问题时最典型的场景。

下面让我们来看一下在C# 1中如何表示空值。

4.1.2　在C# 1 中表示空值的模式

在C# 1中，通常用3种基本的模式来解决“不存在可空值类型”的问题。每种模式都有自己

的优缺点——主要都是缺点——而且所有模式都不太令人满意。但是，仍有必要了解它们。最起码，在了解了它们之后，就会欣赏C# 2集成解决方案的优势。

模式1：魔值

第一种模式是牺牲一个值来表示空值，主要是作为DateTime的解决方案，因为很少有人希望自己的数据库中真的包含公元元年中的某个日期①。也就是说，它有悖于我在前面给出的理由，即假设每个值都能用于一般用途。所以，我们会牺牲一个值（通常是DateTime.MinValue）来表示空值，这个值称为“魔值”。不同的应用程序对它的语义有不同的解释。例如，它可能意味着用户还没有向一个表单中输入值，或者意味着它不适用于当前记录。

使用魔值的一个好处在于，它不会浪费任何内存，也不需要添加任何新的类型。然而，它要求你谨慎选择一个合适值。一经选定，这个值将永远不能用来表示真正的数据。另外，这个设计是不“优雅”的。它让人感觉很“别扭”。如果需要采用这种方式，那么至少应该用一个常量（如果类型不能表示为常量，就要使用静态的、只读的值）来表示魔值，如DateTime.MinValue。不要试图人为地来表达魔值的含义。此外，如果魔值是普通的、有意义的，偶尔用一下也很简单——编译器和运行时都不会帮你找出错误。这里出现的大多数其他方案（包括C# 2的方案），都会因具体情况而产生编译错误或运行时异常。

魔值模式深深地嵌入在IEEE-754二进制浮点类型（如float和double）的计算中。比用单个值表示不是一个真正的数字的想法更进一步——有很多位模式，可以划分为“非数字”（NaN）和正负无穷大。我认为很少有程序员（包括我在内）会给这些值以应有的关注，这又说明了这种模式的缺点。

ADO.NET对这种模式进行了少许修改。在ADO.NET中，所有类型的空值都用魔值DBNull.Value来表示。在这种情况下，是用一个额外的值（实际是一个额外的类型）来表示数据库返回null的情况。但这只适用于编译时的类型安全不重要的情况（也就是说，你愿意使用object，并在检查了是否为空之后进行强制转换。同样，它也让人感觉“别扭”。事实上，它是“魔值”模式和马上要讲到的“引用类型包装”模式的一种混合形式。

模式2：引用类型包装

第二个解决方案可以采取两种形式。较简单的形式是直接用object作为变量类型，并根据需要进行装箱和拆箱。较复杂（同时更吸引人）的形式是假定值类型A可空，就为它准备一个引用类型B。在引用类型B中，包含值类型A的一个实例变量。B中还声明了隐式转换操作符，允许将B转换成A，以及将A转换成B。使用泛型的话，可以在一个泛型类型中完成这一切——但是，如果你已经在使用C# 2，就完全可以换用本章讨论的可空类型。如果只能使用C# 1，就必须为希望包装的每种类型都创建额外的代码。虽然不难以模板的形式来实现代码的自动生成，但这无论如何都是一种“负担”，应尽可能避免。

这两种形式都存在一个问题：虽然它们允许直接使用null，但都要求在堆上创建对象。所以，如果非常频繁地使用这种方式，会造成难以进行垃圾回收。而且伴随着对象所产生的开销，

① 指后文提到的DateTime.MinValue对应的年份。——译者注

内存的消耗也会增大。在较复杂的解决方案中，可以使引用类型成为“易变”[①]类型，这虽然能减少创建实例的数量，但同时也会导致一些非常不直观的代码产生。

模式3：额外的布尔标志

最后一种模式的基本思路是使用一个普通的值类型的值，同时用另一个值（一个布尔标志）来表示值是“真正”存在，还是应该被忽略。同样，有两种方式来实现这个解决方案。要么在代码中维护两个单独的变量，要么将“值和标志”封装到另一个值类型中。

第二种方案与上一节描述的较复杂引用类型的方案非常相似。区别在于，由于使用的是值类型，所以能避免垃圾回收的问题。另外，现在是在封装的值内表示可空性，而不是通过空引用来表示。不过，这种方式存在相同的缺点：针对想要处理的每个值类型，都必须创建一个新的类型。另外，如果值因某种原因要进行装箱，那么不管它是否被认为是空值，都要像平时那样进行装箱。

最后一种模式（采用封装方式）实际就是C# 2的可空类型的工作方式。以后会看到，只要将框架、CLR和语言的新特性结 合到一起，那么与C# 1的各种可选的方案相比，C# 2的方案要简洁得多。下一节将只讨论由框架和CLR提供的支持：如果C# 2只支持泛型，4.2节的大部分内容也无需修改，而且这个特性仍然可用并且很有用，但是C# 2语言提供了更好用的语法糖，详情请参见4.3节。

4.2　System.Nullable<T>和 System.Nullable

可空类型的核心部分是System.Nullable<T>。除此之外，静态类System.Nullable提供了一些工具方法，可以简化可空类型的使用。（从现在起，我会省略命名空间，以方便讨论。）下面将依次讨论这两个类型。另外在本节中，将不会涉及由语言本身提供的额外的特性。这样一来，等到开始讲述C# 2提供的简略表达时，你就能理解IL代码中实际发生的事情。

4.2.1　Nullable<T>简介

从名字就可以看出，Nullable<T>是一个泛型类型。类型参数T有一个值类型约束，所以不能使用Nullable<Stream>。如3.3.1节所述，还意味着不能使用另一个可空类型作为实参。例如，Nullable<Nullable<int>>是不允许的，即使Nullable<T>在其他方面符合值类型的一切特征。对于任何具体的可空类型来说，T的类型称为可空类型的*基础类型*（underlying type）。例如，Nullable<int>的基础类型就是in。

Nullable<T>最重要的部分就是它的属性，即HasValue和Value。它们的作用显而易见：如果存在一个非可空的值（按照以前的说法，就是“真正”的值），那么Value表示的就是这个值。如果（概念上）不存在真正的值，就会抛出一个InvalidOperationException。而HasValue是一个简单的Boolean属性，它指出是存在一个真正的值，还是应该将实例视为null。下面我

① 为可空的值类型创建的这个引用类型本来应该是“不易变”的，对引用类型的实例进行操作，实际会返回新的实例。——译者注

们将讨论“一个有值的实例”（instance with a value）和“没有值的实例”（instance without a value），这两种实例的HasValue属性将分别返回true或false。

可空类型用简单的字段来显式支持这些属性。图4-1展示了Nullable<int>的几个示例，（从左向右）分别表示没有值、0和5。记住，Nullable<T>仍然为值类型，因此对于Nullable<int>类型的变量来说，其值将直接包含一个bool和一个int，而不会是其他对象的引用。

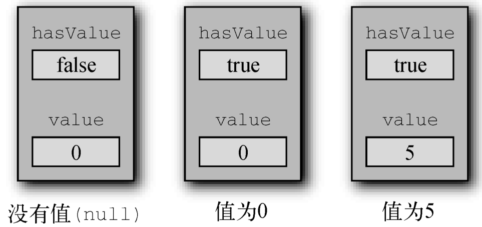


图4-1　Nullable<int>的示例值

既然我们知道了属性的作用，接着来看看如何创建类型的一个实例。Nullable<T>有两个构造函数。其中，默认构造函数创建“一个没有值的实例”。另一个构造函数则接受T的一个实例作为值。实例一经创建，就是“不易变”的。

> **值类型和易变性**
>
> 假如一个类型的实例在创建之后便不能更改，就说这种类型是不易变的。相比跟踪可能会更改共享值的行为来说，不易变类型的设计往往更加干净——特别是在不同线程中时。
>
> “不易变”性对于值类型来说尤其重要。一般说来，值类型应该几乎总是“不易变”的。框架中的大多数值类型是不易变的，但有两个常见的例外——特别是，用于 Windows Forms 和 Windows Presentation Foundation 的 Point 结构是易变的。
>
> 如果需要在一个值的基础上获得另一个值，请按照与 DateTime 和 TimeSpan 一样的思路来操作——提供的方法和运算符应该返回新值，而不是修改现有的值。这避免了各种微妙的bug，比如你想进行修改，但实际上修改的只是副本。对易变值类型说“不”吧。

Nullable<T>引入了一个名为GetValueOrDefault的新方法，它有两个重载方法，如果实例存在值，就返回该值，否则返回一个默认值。其中一个重载方法没有任何参数（在这种情况下会使用基础类型的泛型默认值），另一个重载方法则允许你指定要返回的默认值。

Nullable<T>实现的其他方法全都覆盖了现有的方法：GetHashCode、ToString和Equals。GetHashCode会在实例没有值的时候返回0；如果有值，就返回那个值的GetHashCode。ToString在没有值的时候返回空字符串，否则返回那个值的ToString。Equals稍复杂一些——以后讨论装箱时会谈到它。

最后，框架提供了两个转换。首先，是T到Nullable<T>的隐式转换。转换结果为一个HasValue属性为true的实例。同样，Nullable<T>可以显式转换为T，其作用与Value属性相同，在没有真正的值可供返回时将抛出一个异常。

> **说明　包装（wrapping）和拆包（unwrapping）**　将T的实例转换成Nullable<T>的实例的过程在C#语言规范中称为包装，相反的过程则称为拆包。在C#语言规范中，分别是在涉及“接受一个参数的构造函数”[①]和Value属性时定义的这两个术语。这些调用实际是由C#代码生成的，即使看起来你似乎使用的是由框架提供的转换。但是，无论谁提供转换，结果都是相同的。本章剩余部分不会对这两种实现进行区分。

在进行更深入的讨论之前，先来看看实际的使用。代码清单4-1演示了能直接用Nullable<T>做的事情，暂时把Equals放到一边。

代码清单4-1　使用Nullable<T>的各个成员

```
static void Display(Nullable<int> x)                    ❶ 显示诊断结果
{
   Console.WriteLine("HasValue: {0}", x.HasValue);
   if (x.HasValue)
   {
      Console.WriteLine("Value: {0}", x.Value);
      Console.WriteLine("Explicit conversion: {0}", (int)x);
   }
   Console.WriteLine("GetValueOrDefault(): {0}",
                     x.GetValueOrDefault());
   Console.WriteLine("GetValueOrDefault(10): {0}",
                     x.GetValueOrDefault(10));
   Console.WriteLine("ToString(): \"{0}\"", x.ToString());
   Console.WriteLine("GetHashCode(): {0}", x.GetHashCode());
   Console.WriteLine();
}
...
Nullable<int> x = 5;                                    ❷ 包装等于5的值
x = new Nullable<int>(5);
Console.WriteLine("Instance with value:");
Display(x);

x = new Nullable<int>();                                ❸ 构造没有值的实例
Console.WriteLine("Instance without value:");
Display(x);
```

代码清单4-1首先展示了包装基础类型❷的值的两种不同方式（只是C#源代码不同），然后在实例上使用各种不同的成员❶。接着，创建一个没有值❸的实例，并按相同的顺序使用相同的成员，只是省略了Value属性以及向int的显式转换，因为这些会抛出异常[②]。

代码清单4-1的输出如下所示：

```
Instance with value:
HasValue: True
Value: 5
```

① 前面说过，总共有两个构造函数，一个是默认的，另一个要获取一个参数。——译者注

② 在上面的Display方法中，是用一个if结构来判断是否有值，如果没有值，就不显示Value属性的值，也不执行向int的显式转换。——译者注

```
Explicit conversion: 5
GetValueOrDefault(): 5
GetValueOrDefault(10): 5
ToString(): "5"
GetHashCode(): 5

Instance without value:
HasValue: False
GetValueOrDefault(): 0
GetValueOrDefault(10): 10
ToString(): ""
GetHashCode(): 0
```

到目前为止，所有结果都在预料之中，只需看一看Nullable<T>提供的成员，就一切都明白了。但是，在涉及装箱和拆箱时，会有一些特殊的行为出现。这会使可空类型的行为符合我们真正的想法，普通的装箱规则再也不能捆住我们的手脚。

4.2.2 Nullable<T>装箱和拆箱

请牢记，Nullable<T>是一个结构——一个值类型！这意味着如果把它转换成引用类型（如object是），就要对它进行装箱。只有在涉及装箱和拆箱时，CLR才会让可空类型有一些特殊的行为。其他时候，可空类型使用的是“标准”的泛型、转换、方法调用，等等。事实上，这个行为是在.NET 2.0发布前不久才修订的，是程序员社区强烈要求的结果。在预览版中，可空值类型的装箱方式与其他值类型相同。

Nullable<T>的实例要么装箱成空引用（如果没有值），要么装箱成T的一个已装箱的值（如果有值）。参见图4-2，它永远不可能装箱成“装箱的可空int”，因为不存在这种类型。

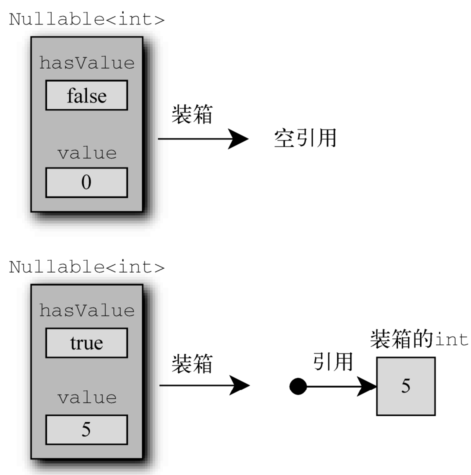


图4-2 没有值的实例的装箱结果（上）和有值的实例的装箱结果（下）

已装箱的值可以拆箱成普通类型，或者拆箱成对应的可空类型[①]。拆箱一个空引用时，如果

① 换言之，要么拆箱成T，要么拆箱成Nullable<T>。——译者注

拆箱成普通类型，会抛出一个NullReferenceException；但如果拆箱成恰当的可空类型，就会拆箱成没有值的一个实例。代码清单4-2展示了这个行为。

代码清单4-2　可空类型的装箱和拆箱行为

```
Nullable<int> nullable = 5;

object boxed = nullable;                         装箱成“有值的可空类型的实例”
Console.WriteLine(boxed.GetType());

int normal = (int)boxed;                         拆箱成非可空变量
Console.WriteLine(normal);

nullable = (Nullable<int>)boxed;                 拆箱成可空变量
Console.WriteLine(nullable);

nullable = new Nullable<int>();
boxed = nullable;                                装箱成“没有值的可空类型的实例”

Console.WriteLine(boxed == null);

nullable = (Nullable<int>)boxed;                 拆箱成可空变量
Console.WriteLine(nullable.HasValue);
```

从代码清单4-2的输出可以看出，在输出已装箱的值的类型时，显示的是System.Int32，而不是System.Nullable<System.Int32>。随后，程序证明了为了获取值，既可以拆箱成int，也可以拆箱成Nullable<int>。最后，我们将“一个没有值的可空实例”装箱成一个空引用，然后成功拆箱成另一个没有值的可空实例。如果企图将boxed的最后一个值拆箱成非可空的int，程序会抛出一个NullReferenceException。

理解了装箱和拆箱的行为之后，就可以开始研究Nullable<T>.Equals的行为了。

4.2.3　Nullable<T>实例的相等性

Nullable<T>覆盖了object.Equals(object)，但没有引入任何相等性操作符，也没有提供Equals(Nullable<T>)方法。由于框架已经提供了基础的架构，所以语言能在其顶部添加额外的功能，包括使现有的操作符能像我们希望的那样工作。详细内容将在4.3.3节讲述，这里只是讲一下由Equals方法定义的基本的相等性。调用first.Equals(second)的具体规则如下。

- 如果first没有值，second为null，它们就是相等的；
- 如果first没有值，second不为null，它们就是不相等的；
- 如果first有值，second为null，它们就是不相等的；
- 否则，如果first的值等于second，它们就是相等的。

注意，我们不必考虑second是另一个Nullable<T>的情况，因为装箱规则杜绝了此类情况的发生。second的类型是object，所以如果要成为Nullable<T>，它就必须进行装箱。而前面提到，对一个可空实例进行装箱，会创建非可空类型的一个“箱子”，或者返回一个空引用。第一条规则似乎违反了Object.Equals(object)的一个契约，即x.Equals(null)会返回false——但实际上只有x为非空引用时才会返回false。同样，由于装箱行为的存在，永远不会通过引用调用

Nullable<T>的实现。

这些规则与.NET其他地方的相等性规则是一致的。所以，可空实例可以作为字典的键来使用。另外，在需要相等性的其他所有情况下，都可以使用它。不要指望能对“一个非可空实例”和“有值的可空实例”这两种情况进行区分——一切都已进行了精心设计，所以对这两种情况的处理方式是相同的。

对Nullable<T>结构本身的讨论就是这么多。接着来看看和一个它关系密切的类：Nullable。

4.2.4　来自非泛型**Nullable**类的支持

System.Nullable<T>结构能做你希望它做的几乎一切事情。不过，它也得到了System.Nullable类的帮助。这是一个静态类，它只包含了静态方法，不能创建它的实例[①]。事实上，它能完成的一切操作都能用其他类型很好地完成。如果微软一开始就考虑周全，它根本就不应该存在——这样一来，我们就不会对为什么存在这两种类型产生疑惑。由于历史原因遗留下来的Nullable类提供了3个方法，它们至今仍然有用。

前两个方法是比较方法：

```
public static int Compare<T>(Nullable<T> n1, Nullable<T> n2)
public static bool Equals<T>(Nullable<T> n1, Nullable<T> n2)
```

Compare使用Comparer<T>.Default来比较两个基础值（如果它们存在的话），Equals使用EqualityComparer<T>.Default。对于没有值的实例，上述每个方法返回的值都遵从.NET的约定：空值与空值相等，小于其他所有值。

这两个方法其实完全可以设计成Nullable<T>的静态、非泛型成员方法。现在把它们设计成一个非泛型类型中的泛型方法，唯一的好处就是可以应用泛型类型推断，这样就不需要显式指定类型参数了。

System.Nullable的第3个方法不是泛型的——也绝对不可能是！其签名如下：

```
public static Type GetUnderlyingType(Type nullableType)
```

如果参数是一个可空类型，方法就返回它的基础类型；否则就返回null。之所以不可能是泛型方法，是由于假如一开始就知道基础类型是什么，就没有必要调用这个方法了。

现在，我们已经知道了框架和CLR为可空类型提供的支持。接下来看看C# 2为了简化可空类型的使用而引入的语言特性。

4.3　C# 2 为可空类型提供的语法糖

到目前为止列举的所有例子，虽然确实证明了可空类型能很好地发挥作用，但它们不能令人十分满意。在你输入Nullable<>时，很显然你打算使用可空类型。但是，这种写法强调了可空

① 第7章会更多地讨论静态类。

性，却使基础类型的名称变得好像无足轻重。这显然不是个好主意。

除此之外，“Nullable”这个名称暗示着应该能将`null`赋给可空类型的变量。但我们并没有看到这一点——我们一直使用的都是类型的默认构造函数。本节将探讨C# 2如何处理此类问题。

在深入探讨C# 2语言提供的细节之前，有一个定义终于可以正式和大家见面了。可空值类型的空值（null value）是指在“`HasValue`返回`false`”时的值。或者是“实例没有值”时的值，就像4.2节指出的那样。之前之所以没有采用它，是因为它是C#特有的。CLI规范和`Nullable<T>`本身的文档都没有提到它。所以，我一直等到讨论C# 2语言本身时，才引入对空值的定义。这个定义也适用于引用类型：引用类型的空值是指空引用，我们在C# 1中已经很熟悉了。

说明　**可空类型与可空值类型**　在C#语言规范中，**可空类型**是指可以包含空值的类型——如引用类型和`Nullable<T>`。你应该已经注意到了，我一直在使用可空类型替换**可空值类型**（显然不包括引用类型）。尽管在术语方面我通常都一丝不苟，但如果本章到处充斥着“可空值类型”，读起来实在是太费劲了。实际代码中“可空类型”的使用也有些模糊：它更经常用于描述`Nullable<T>`，而不是像规范中规定的那样。

知道空值在C#中的含义之后，来看看C# 2为我们提供了什么特性。先看看它如何对代码进行精简。

4.3.1　?修饰符

一些语法元素虽然刚开始不太熟悉，但却能让人恰当地感知它们的含义。条件操作符（`a ? b : c`）对我来说就是其中之一——它先问了一个问题（`a`），然后有两个对应的回答（`b`和`c`）。同样，用于可空类型的?修饰符我也能猜到它的用途。

它是指定可空类型的一种快捷方式。所以，现在不需要写`Nullable<byte>`，写`byte?`就可以了。两者可以互换使用，最终会编译成完全相同的IL，所以，你可以同时使用这两种写法。当然，考虑到还有其他人会读你的代码，所以应该一开始就确定自己想采用的写法，并坚持采用这种写法。代码清单4-3用?修饰符改写了代码清单4-2，改写部分用粗体表示，其实这两段代码的功能完全相同。

代码清单4-3　使用?修饰符来改写代码清单4-2

```
int? nullable = 5;

object boxed = nullable;
Console.WriteLine(boxed.GetType());

int normal = (int)boxed;
Console.WriteLine(normal);

nullable = (int?)boxed;
Console.WriteLine(nullable);

nullable = new int?();
```

```
boxed = nullable;
Console.WriteLine(boxed == null);

nullable = (int?)boxed;
Console.WriteLine(nullable.HasValue);
```

至于这些代码都做了什么事情，以及具体是如何实现的，这里就不再赘述了，因为结果和代码清单4-2完全一样。两个代码清单编译成相同的IL——它们只是使用了不同的语法，就像int和System.Int32可以换着用一样。在代码清单4-3中，所有发生改变的地方都进行了加粗强调。你可以在任何地方使用这种简写方式，包括在方法签名中，在typeof表达式中，以及在强制类型转换中，等等。

4

我之所以觉得这个修饰符选得非常好，是因为它让人对变量的本质产生了一种不确定的感觉。你会想代码清单4-3中的nullable有一个整数值吗？在任何特定的时刻，它都可能是一个整数值，也可能是空值。

从现在起，我们的所有例子都会使用?修饰符，因为这种写法更简洁，而且是在C#中使用可空类型的标准方式。然而，有人会认为，在阅读代码时，会很容易漏看这个修饰符。如果你也这样想，那么完全可以使用较长的语法。可以比较一下代码清单4-3和代码清单4-2，自己判断哪一个更清晰。

由于C# 2语言规范中已定义了空值，如果不能使用语言中已经存在的null字面量来表示空值，会让人感觉很奇怪。幸好我们能做到。

4.3.2　使用**null**进行赋值和比较

如果图省事，用一句话就可以概括本节的内容："C#编译器允许使用null在比较和赋值时表示一个可空类型的空值。"但我不愿意这么"省事"。相反，我希望用真实的代码来演示这句话的含义，并指导你思考为什么语言提供了这一特性。

每次使用Nullable<T>的默认构造函数时，你都可能有一点"怪怪"的感觉。它能产生需要的行为，但不能解释我们这样做的原因。所以它没有给读者留下应有的印象。我们希望读者获得和为引用类型使用null时相同的感受。

读到这里，有人可能会产生疑问，在这一节和上一节，我都讲到了"感觉"。是的，你只需想一想是谁在写代码，又是谁在读代码，就知道"感觉"的重要性了。诚然，编译器是必须理解代码的，它对写法的微妙之处可能并不怎么关心。但是，对于生产系统中使用的代码，很少存在写了之后就再也没人去读的情况。因此，必须尽一切可能使读代码的人跟上你当初写代码时的思路——使用熟悉的null字面量，可以帮助我们实现这一点。

以前的例子实际只是简单地演示了语法和行为。后面的例子准备进行一些变化，让读者真正理解可空类型的使用方式。我们考虑建立一个Person类，你需要知道人的姓名、出生日期和死亡日期。我们记录的人肯定是已经出生的，而且其中一些人仍然健在，在这种情况下，死亡日期就要用null来表示。代码清单4-4展示了一部分代码。这个例子只关注年龄的计算，但在一个真正的Person类中，肯定还应该包含更多操作。

代码清单4-4　Person类的部分代码，其中包含了年龄的计算

```
class Person
{
   DateTime birth;
   DateTime? death;
   string name;

   public TimeSpan Age
   {
      get
      {
         if (death == null)                  ❶ 检查HasValue
         {
            return DateTime.Now - birth;
         }
         else
         {
            return death.Value - birth;            ❷ 拆包以进行计算
         }
      }
   }

   public Person(string name,
                 DateTime birth,
                 DateTime? death)
   {
      this.birth = birth;
      this.death = death;
      this.name = name;
   }
}
...
Person turing = new Person("Alan Turing ",
                           new DateTime(1912, 6, 23),      ❸ 包装成可空实例
                           new DateTime(1954, 6, 7));
Person knuth = new Person("Donald Knuth ",
                          new DateTime(1938, 1, 10),     ❹ 指定死亡日期为null
                          null);
```

代码清单4-4没有产生任何输出，但假如你之前没有读过本章，仅仅是它能通过编译这一事实，就可能让你感到很“震撼”。那么除了?修饰符外，DateTime?居然能和null进行比较，并且能将null作为DateTime?参数的实参来传递，同样会让你觉得非常奇怪。

不过现在，它们的含义很直观——将death变量同null进行比较时，是在问它的值是否为空值。同样，将null作为DateTime?实例来使用时，实际是通过调用类型的默认构造函数为这个类型创建空值。研究一下生成的IL代码，就会发现在编译器为代码清单4-4生成的代码中，实际是调用了death.HasValue属性❶，并使用默认构造函数（在IL中用initobj指令来表示）来创建DateTime?的新实例❹。为了创建Alan Turing（阿兰·图灵）的死亡日期❸，代码调用普通的DateTime构造函数，并将结果传给有一个参数的Nullable<DateTime>构造函数。

之所以让你查看IL，是因为这是查看代码实际所做的工作的一种非常有用的方式，尤其是在

你觉得编译结果和你希望的结果不符时。可以使用与.NET SDK提供的ildasm工具来查看IL，也可以使用.NET Reflector、ILSpy、dotPeek或者JustDecompile等反编译器来查看。（我习惯使用Reflector，所以在本书中总是提到它，但其他几个工具也非常好用。）

前面描述了C#为“空值”这一概念提供的简化语法。只要人们开始理解了可空类型的概念，这种写法就能极大地改善代码的“表现力”。然而，代码清单4-4有一部分代码所做的工作要比我们希望的多，也就是用减法运算来求一个已故的人活了多少岁的那一步❷。为什么一定要对值进行拆包处理呢？为什么不能直接返回`death-birth`呢？在`death`为`null`的情况下，我们希望表达式表达出什么意思（当然要排除在代码清单4-4中最开始判断是否为`null`的情况）？这些问题（以及其他许多问题）将在下一节得到解答。

4.3.3 可空转换和操作符

前面已经说过，可空类型的实例能与`null`进行比较，但在某些情况下，还有可能要进行另一些比较，而且可能要使用其他操作符。同样，前面虽然已经讲到包装和拆包，但一些类型还有可能进行其他转换。本节将解释这些可进行的操作。虽然像这样的主题听起来也许会让人感觉乏味，但正是这些经过了精心设计的特性，才使得C#成为一种你可以长期使用的语言。刚开始时不能全部听懂也没有关系：只需记住细节已经在这节提供了，等你以后在编程中用到时，随时都可以回到这里来重新阅读并加深理解。

简单地说，假如一个非可空的值类型支持一个操作符或者一种转换，而且那个操作符或者转换只涉及其他非可空的值类型时，那么可空的值类型也支持相同的操作符或转换，并且通常是将非可空的值类型转换成它们的可空等价物。下面举一个更具体的例子。我们知道，`int`到`long`存在着一个隐式转换，这就意味着`int?`到`long?`也存在一个隐式转换，其行为可想而知。

可惜，虽然这种泛泛而谈的描述能使人获得一个大致的印象，但具体的规则要稍微复杂一些。每条规则都很简单，但规则的数量太多了。你有必要知道这些规则，否则极有可能在遇到一个编译器错误或者警告时一头雾水，不知道为什么编译器认为你要执行一个你压根儿没打算执行的转换。我们将先讨论转换，再讨论操作符。

1. 涉及可空类型的转换

为了保持内容的完整性，先来看一下我们已经知道的转换：

- `null`到`T?`的隐式转换；
- `T`到`T?`的隐式转换；
- `T?`到`T`的显式转换。

现在来看看类型所支持的预定义转换和用户自定义转换。例如，`int`到`long`存在一个预定义转换。假如允许从非可空值类型（`S`）转换成另一个非可空值类型（`T`），那么同时允许进行以下转换：

- `S?`到`T?`（可能是显式或隐式的，具体取决于原始转换）；
- `S`到`T?`（可能是显式或隐式的，具体取决于原始转换）；
- `S?`到`T`（总是显式的）。

还是使用前面的例子，这意味着可以隐式地从int?转换为long?，隐式地从int转换为long?，以及显式地从int?转换为long。转换过程很自然，S?的空值会转换为T?的空值。如果转换的是非空的值，就使用原始转换。和往常一样，如果S?是空值，那么从S?到T的显式转换会抛出一个InvalidOperationException。对于用户自定义的转换，这些涉及可空类型的额外转换称为*提升转换*（lifted conversion）[①]。

到目前为止，一切都十分简单。接着来研究一下操作符，它们的情况要稍微复杂一些。

2. 涉及可空类型的操作符

C#允许重载以下操作符：

- 一元：+ ++ - -- ! ~ truefalse
- 二元：+ - * / % & | ^ << >>
- 相等：[②]== !=
- 关系：< > <= >=

当为非可空的值类型T重载了上述操作符之后，可空类型T?将自动拥有相同的操作符，只是操作数和结果的类型稍有不同。这些操作符称为*提升操作符*[③]——不管它们是预定义的操作符（比如对数值类型执行加法运算），还是用户自定义的操作符（比如将一个TimeSpan加到一个DateTime上）。提升操作符在使用时存在着一些限制：

- true和false操作符永远不会被提升，这两个操作符本身就十分少用，所以这个限制对我们的影响不大；
- 只有操作数是非可空值类型的操作符才会被提升；
- 对于一元和二元操作符（相等和关系操作符除外），返回类型必须是一个非可空的值类型；
- 对于相等和关系操作符，返回类型则必须是bool；
- 应用于bool?的&和|操作符有单独定义的行为，4.3.4节将介绍。

对于所有操作符，操作数的类型都成为它们的可空等价物。对于一元和二元操作符，返回类型也成为可空类型。如果任何一个操作数是空值，就返回一个空值。相等和关系操作符的返回类型为非可空布尔型。进行相等测试时，两个空值被认为相等，空值和任何非空值被认为不等。这和4.2.3节描述的行为一致。对于关系操作符，如果它的任何一个操作数是空值，那么返回的始终是false。如果没有任何操作数是空值，那么自然应该使用非可空类型的操作符。

所有这些规则听起来比较复杂，但实际操作起来并不难——基本上，一切都会像你期望的那样工作。用几个例子可以很容易地说明发生的事情，由于int具有如此多的预定义操作符（并且整数也很好表示），所以用它进行演示是再自然不过的了。表4-1演示了多个表达式、经提升的操作符签名以及结果。它假定存在变量four、five和nullInt，每个变量的类型都是int?，它们的值不言自明。

① 有关提升转换的详情，请参见C# 4.0规范的6.4节。——译者注
② 相等和关系操作符本身自然是二元操作符。但是，由于它们的行为和其他二元操作符稍有区别，所以单独分为一类。
③ 有关提升操作符的详情，请参见C# 4.0规范的7.2.7节。——译者注

表4-1　向可空整数应用提升操作符的例子

表　达　式	经提升的操作符	结　　果
-nullInt	int? –(int? x)	null
-five	int? –(int? x)	-5
five + nullInt	int? +(int? x, int? y)	null
five + five	int? +(int? x, int? y)	10
nullInt == nullInt	bool ==(int? x, int? y)	true
five == five	bool ==(int? x, int? y)	true
five == nullInt	bool ==(int? x, int? y)	false
five == four	bool ==(int? x, int? y)	false
four < five	bool <(int? x, int? y)	true
nullInt < five	bool <(int? x, int? y)	false
five < nullInt	bool <(int? x, int? y)	false
nullInt < nullInt	bool <(int? x, int? y)	false
nullInt <= nullInt	bool <=(int? x, int? y)	false

在这个表格中，恐怕最让人不解的就是最后一行，尽管这两个空值真的相等（如第5行所示），但却不能认为一个空值“小于或等于”另一个空值。是不是很奇怪？但根据我的经验，这个“矛盾”在实际应用中并不会造成任何问题。

提升操作符和可空转换可能造成混淆的一个地方在于：当你使用一个非可空的值类型时，可能会无意间同null进行比较。以下代码虽然是合法的，但没有什么实际用处：

```
int i = 5;
if (i == null)
{
   Console.WriteLine ("Never going to happen");
}
```

C#编译器会为上述代码显示一个警告，但你或许会很惊讶，它居然能通过编译。这里发生的事情是：编译器看到了==左侧的int表达式，又看到了右侧的null，并知道两者都存在到int?的一个隐式转换。由于对两个int?值进行比较是完全有效的，所以代码不会产生错误——而只是警告[①]。更复杂一些的一个问题是，如果不是int，而是一个被约束为值类型的泛型类型参数，像这样的比较就是非法的——在这种情况下，泛型规则禁止与null进行比较。

无论如何，要么报错，要么至少有一个警告，所以假如你有仔细阅读警告消息的好习惯，最终的代码就应该能避免这个问题。另外，由于这里提前指出了可能遇到的问题，所以等到你真正遇到这个问题时，就不会为此感到头疼了。

现在可以回答上一节最后提出的问题：为什么在代码清单4-4中要使用death.Value- birth，而不是直接使用death-birth。根据前面描述的规则，death-birth这个表达式是可以使用的，但结果就会是一个TimeSpan?，而不是一个TimeSpan。这样一来，就有两种选择：将结果强制转换为TimeSpan，使用它的Value属性，或者更改Age属性以返回一个TimeSpan?——这只是将问题转嫁到了调用者的身上。4.3.6节会给出Age属性的一个更好的实现（虽然仍然不是最完

① 编译器显示的警告正文是：“由于int类型的值永不等于int?类型的null，所以该表达式的结果始终为false。”
——译者注

美的）。

在与操作符提升有关的限制中，我提到过`bool?`的工作方式稍微有别于其他类型。下一节会解释原因，并站在更加全局的角度重点分析这些操作符为什么会按现在的方式工作。

4.3.4　可空逻辑

我清楚地记得在学校时上电子学课程的情形，那时好像主要做两件事情：要么使用V=I×R这个公式来计算一个电路各个部分的电压，要么就是应用真值表——这个图表解释了NAND①门和NOR门等的差异。真值表的基本思路非常简单——将每一种可能的输入组合都映射为一个逻辑，并告诉你输出是什么。

在那个时候，我们画的真值表比较简单，2个输入的逻辑门总是有4行——每个输入都有2个可能的值，这意味着总共有4种可能的组合。布尔逻辑就是这么简单，但在面对一个三态的逻辑类型时，又会发生什么呢？`bool?`就是这样的一个类型，它的值可能为`true`、`false`或`null`。那意味着由于要使用二元操作符，我们的真值表的行数要增加到9行，因为总共有9种不同的组合。语言规范只强调了逻辑AND和OR操作符（分别是`&`和`|`），这是因为其他操作符（一元逻辑取反操作符`!`和异或操作符`^`）与其他提升操作符遵循同样的规则。没有为`bool?`定义条件逻辑操作符（短路`&&`和`||`符），这让我们省了一点事儿。

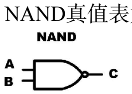
考虑到完整性，表4-2展示了所有4个有效的`bool?`操作符的真值表。

表4-2　逻辑操作符&、|、^和!应用于bool?类型时的真值表

x	y	x & y	x | y	x ^ y	!x
true	true	true	true	false	false
true	false	false	true	true	false
true	null	null	true	null	false
false	true	false	true	true	true
false	false	false	false	false	true
false	null	false	null	null	true
null	true	null	true	null	null
null	false	false	null	null	null
null	null	null	null	null	null

能找出这些规则的基本原理比仅仅查看这张表中的结果值更有助于理解，其基本思路在于：一个为空的`bool?`值从某种意义上来说代表着一种"可能"。就上表来说，可以将输入的每个空项都想象成一个变量，假如结果取决于该变量的值，结果就肯定为`null`。以表中的前3行为例，只有在`y`为`true`时，`true & y`才为`true`。但是，不管`y`的值是什么，`true | y`总是为`true`。由于前者的结果取决于`y`，后者的结果不取决于`y`，所以假如`y`的值是`null`，那么前者的结果就是`null`，后者则为`true`。

① NAND真值表如下图所示：



A	B	C
1	1	0
1	0	1
0	1	1
0	0	1

——译者注

其实C# 1的空引用和SQL的NULL值是稍有抵触的。语言的设计者在决定提升操作符的工作方式，尤其是可空逻辑的工作方式时，必须作出正确的决策。在许多时候，它们根本不会发生冲突——在C# 1中，不存在对空引用应用逻辑操作符的概念。所以，使用早先给出的SQL风格的结果时，不会出现任何问题。但是，在进行比较时，我们的定义可能会使一些SQL开发人员感到惊奇。在标准SQL中，对两个值进行比较时（进行相等性比较，或者进行大于/小于比较时），假如任何一个值是NULL，则结果肯定是未知的。在C# 2中进行这些比较，结果永远不会为null。尤其要注意，两个null值视为相等。

说明 **记住：这是C#特有的！** 有必要记住，本节讨论的提升操作符和转换，还有bool?逻辑，它们都是由C#编译器提供的，而不是由CLR或者框架本身提供的。对使用了任何可空操作符[①]的代码执行ildasm，会发现编译器已经创建了所有恰当的IL来测试空值，并相应地对它们进行处理。这意味着不同的语言在这方面可能有不同的行为。如果需要在不同的基于.NET的语言之间移植代码，这无疑是你应该特别注意的地方。例如，VB看待提升操作符的方式跟SQL非常像，所以如果x或y的空值，那x<y的结果也是空值。

还有一种熟悉的操作符也可以用于可空类型，考虑一下你掌握的空引用知识，并将其应用到空值上，就可以达到自己想要的效果。

4.3.5 对可空类型使用as操作符

在C# 2之前，as操作符只能用于引用类型。而在C# 2中，它也可以用于可空类型。其结果为可空类型的某个值——空值（如果原始引用为错误类型或空）或有意义的值。下面是一个简短的示例：

```
static void PrintValueAsInt32(object o)
{
   int? nullable = o as int?;
   Console.WriteLine(nullable.HasValue ?
                     nullable.Value.ToString() : "null");
}
...
PrintValueAsInt32(5);                  ← 生成5
PrintValueAsInt32("some string");              ← 生成null
```

这样我们仅需一步（当然，通常还需要一些检查）就可以安全地将任意引用转换为值。而在C# 1中，你只能先使用is操作符，然后再强制转换，这会使CLR执行两次相同的类型检查，显然不够优雅。

① 可空操作符原文是nullable operator，这是作者自己发明的一个词，其实就是前面描述过的提升操作符，或者说用于对可空类型进行操作的操作符。——译者注

> **说明　惊人的性能陷阱**　我一直以为进行一次检查要比两次快，但实际上并非如此——至少在我测试的.NET版本（.NET 4.5以及之前的版本）中不是这样。假设有一个存储整数的`object[]`类型数组，其中只有1/3为已装箱的整型。在为这样的数组编写快速基准（benchmark）时我发现，使用`is`和强制转换要比使用`as`操作符快20倍。我们在此不会讨论其细节，因为这超出了本书范围，并且通常在选择最佳策略之前，你一般都会针对特殊的环境使用实际的代码和数据进行性能测试——但这值得注意。

现在，我们已经掌握了足够多的知识来使用可空类型并预测它们的行为。不过C# 2在语法增强方面还附赠了一个特性：空合并操作符。

4.3.6　空合并操作符

到目前为止，除了?修饰符之外，C#编译器处理可空类型时的技巧都是在利用现有的语法。然而，从C# 2开始引入了一个新的操作符，能一定程度上使代码变得更简单明了。它称为空合并（null coalescing）操作符，用两个操作数之间的??符号来表示。它有点儿像条件操作符，只不过专门为空值进行了调整。

这个二元操作符在对`first ?? second`求值时，大致会经历以下步骤：

(1) 对`first`进行求值；

(2) 如结果非空，则该结果就是整个表达式的结果；

(3) 否则求`second`的值，其结果作为整个表达式的结果。

之所以说“大致”，是因为在语言规范的完整描述中，有许多情况都涉及`first`和`second`的类型转换问题。但和以前一样，在操作符的大多数应用中，这些问题都并不重要，所以我不想完全照搬规范。如果需要了解详细信息，请参考7.13节。

重要的是，假如`second`的类型是`first`的基础类型（因此是非可空的），那么最终的结果就是那个基础类型。例如，下面的代码是完全合法的：

```
int? a = 5;
int b = 10;
int c = a ?? b;
```

注意，尽管c是非可空的`int`类型，我们还是直接对它赋值。之所以可以这么做，是因为b是非可空的，因此最终将得到一个非可空的结果。

显然，这个示例非常简单。我们来看一个更实际的用法，再次回到代码清单4-4的`Age`属性。以下是其实现，包含一些相关的变量声明：

```
DateTime birth;
DateTime? death;

public TimeSpan Age
{
   get
   {
```

```
        if (death == null)
        {
           return DateTime.Now - birth;
        }
        else
        {
           return death.Value - birth;
        }
     }
  }
```


注意，if语句的两个分支如何从一个非空的DateTime值中减去birth值的。我们感兴趣的值是健在者的最新统计时间——如果这个人已经死亡，就是他死亡的时间，否则就是now。为了循序渐进，先使用普通的条件操作符：

```
DateTime lastAlive = (death == null ? DateTime.Now : death.Value);
return lastAlive - birth;
```

这貌似算是一个进步，但条件操作符的引入，虽然使新的代码缩短一些，很难说是使代码变得更易读还是更难读了。条件操作符经常如此——用得多还是用得少，这纯属个人喜好。使用它之前，最好先和开发团队的其他成员协商好。现在，我们看看空合并操作符对代码有哪些改进。如果death非空，就使用death的值，否则使用DateTime.Now。所以可以将实现修改成以下形式：

```
DateTime lastAlive = death ?? DateTime.Now;
return lastAlive - birth;
```

注意结果的类型是DateTime而不是DateTime?，因为我们使用DateTime.Now作为第2个操作数。其实可以进一步精简成一个表达式：

```
return (death ?? DateTime.Now) - birth;
```

但这样写不利于理解，尤其是在两行代码的版本中，变量lastAlive能帮助阅读代码的人理解为什么要应用空合并操作符。我希望你同意这个说法：与使用if语句的原始版本相比，或者与使用C# 1普通条件操作符的版本相比，两行代码的版本显得更简单、更易读。当然，前提是读者理解空合并操作符的工作原理。根据我个人的经验，这应该是C# 2最不为人所知的一个设计。但是，由于它是如此有用，所以完全值得说服你的同事使用它，而不是故意去避开它。

??操作符的另外两个特性增强了它的可用性。首先，它并非只能用于可空值类型，还能应用于引用类型。你只是不能用非可空值类型作为第一个操作数，因为那就完全没有意义了。另外，它的结合性是*右结合*（right associative）。这意味着表达式first ?? second ?? third实际相当于first ?? (second ?? third)。如果还有更多的操作数，可以此类推。可以使用任意数量的表达式，并依次对它们求值，遇到第一个非空的结果就停止。如果所有表达式的值都为空，则结果也为空。

举一个更具体的例子，假定你有一个在线订购系统，它有billing address（账单寄送地址）、contact address（联系地址）以及shipping address（送货地址）等概念。业务规则规定，任何用户都必须有一个billing address，但contact address是可选的。对一个特定的订单来说，送货地址也是可选的，它默认为billing address。这些“可选”地址在代码中可以很容易地表示为空引用。遇到

送货问题时，为了找到联系人，C# 1的代码可能会这样写：

```
Address contact = user.ContactAddress;
if (contact == null)
{
   contact = order.ShippingAddress;
   if (contact == null)
   {
      contact = user.BillingAddress;
   }
}
```

如果在这种情况下使用条件操作符，会使代码变得很可怕。然而，如果使用??操作符，代码就会变得相当直观易懂：

```
Address contact = user.ContactAddress ??
                  order.ShippingAddress ??
                  user.BillingAddress;
```

如果以后业务规则发生了改变，要默认使用shipping address而不是用户的contact address，那么在上述代码中应该如何修改显而易见。虽然if/else版本修改起来也不是特别麻烦，但我必须三思而后行。修改完毕后，必须在大脑中“过”一遍才能放心。当然，还可以依赖于单元测试，所以我犯错的几率可以说是相当小的。但是，除非有绝对的必要，否则我不愿意这样大费周折。

说明　一切都要适可而止　可能有的人会想，代码中会不会因为出现大量??操作符而显得凌乱不堪，这实际是不会发生的。当我发现默认机制中涉及空值，可能要使用条件操作符时，才会考虑使用??操作符，但这种情况并不多见。但无论如何，只要正常使用这个操作，就会大大增强可读性。

前面讲了如何将可空类型用于对象的“普通”属性——这些属性表示为值类型，但在某些时候可能没有值。这是可空类型最明显的一种用法，也确实是最常见的一种用法。但是，还有其他几个模式虽然不是特别常见，但假如熟练使用，仍然能发挥出强大的作用。下一节将探讨其中的两种模式。这主要是出于兴趣，而不完全是为了了解可空类型本身的行为，才学习这部分内容。现在，你已经掌握了在代码中使用可空类型时所需的全部工具。但是，如果你对一些奇怪的想法感兴趣，并想尝试一些新的东西，请继续阅读……

4.4　可空类型的新奇用法

在可空类型出现之前，我注意到有许多人都很需要它，而且一般都和数据库访问有关。但是，数据库访问并不是可空类型的唯一用途。本节描述的使用模式有一点点“非传统”，但确实能使代码变得更简单。如果你只想“正常”地使用C#，那完全无可厚非——本节可能不适合你，我完全同意这种观点。一般情况下，相比“聪明”的代码，我更喜欢简单的代码，但假如一个完整的模式能够带来好处，那么有时也值得学习它。当然，是否使用这些技术完全取决于你，但你会发

现，这些技术背后的思想可能会对你代码中其他部分的设计有所启示。

不多说了，先来看看3.3.3节提到的TryXXX模式的备选方案。

4.4.1 尝试一个不使用输出参数的操作

用一个返回值来判断一个操作是否成功，并用一个输出参数来返回真正的结果，这种模式在.NET Framework中变得越来越常用。它的目的是合理的——有的方法确实可能在一些非异常的情况下无法实现最初的目标。但现在的问题是，我并不是特别喜欢使用输出参数。在一行中声明一个变量，并立即把它作为输出参数使用时，其语法略显笨拙。

返回引用类型的方法经常会使用这样一种模式：失败时返回null，成功则返回一个非空的值。但是，假如在方法执行成功的前提下，null也是一个有效的返回值，再这样做就不起作用了。Hashtable很好地印证了上面这两句话，只是采取的是一种稍显矛盾的方式。你看到了，从理论上说，null应该是Hashtable中的一个有效值。但是，我见过大多数使用Hashtable的情况都从来不用null值。这意味着完全可以在代码中假定null值代表一个缺失的键。

一种常见的情形是让Hashtable的每个值都成为一个列表：为某个特定的键首次添加一项时，就创建一个新的列表，并将项添加到这个列表中。之后，再为此键添加另一个项时，会将这个项添加到现有的列表中。以下是C# 1中的代码：

```
ArrayList list = hash[key];
if (list == null)
{
   list = new ArrayList();
   hash[key] = list;
}
list.Add(newItem);
```

希望你使用更符合自己实际情况的变量名，不过我相信你已理解了它的思路，并可能已经很好地使用过它[①]。使用可空类型，这种模式可以扩展至值类型——而且事实上，它用于值类型时更安全，因为假如自然的结果类型是值类型，那么一旦返回null，就表示出错了。可空类型以语言支持的泛型方式来添加了额外的布尔信息——何不用它们呢？

为了演示该模式在除了“字典查找”之外其他地方的实际运用，我将使用TryXXX模式的经典例子——解析一个整数。在代码清单4-5中，TryParse方法的实现演示了这个模式使用输出参数的版本。随后，在程序的Main[②]部分，我演示了使用可空类型的版本。

代码清单4-5 TryXXX模式的备选实现

```
static int? TryParse(string text)
{
```

① 假如Hashtable和Dictionary<TKey, TValue>可以获取一个委托，每次在查找到一个不存在的键时就调用这个委托来添加一个新的值，这样是不是很方便？使用这个模式，像这样的情形可以变得非常简单。

② 如1.4节所述，本书的代码段都是用作者自己开发的Snippy程序来加载、编译和运行的。省略号之后的就是程序的Main部分。——译者注

```
    int ret;
    if (int.TryParse(text, out ret))          使用输出参数的经典调用
    {
       return ret;
    }
    else
    {
       return null;
    }
}
...
int? parsed = TryParse("Not valid");          可空调用
if (parsed != null)
{
   Console.WriteLine ("Parsed to {0}", parsed.Value);
}
else
{
   Console.WriteLine ("Couldn't parse");
}
```

你可能会觉得两个版本很难区分，毕竟它们的行数是一样的。但是，我认为有一个重要的区别。可空版本将自然的返回值和解析是否成功封装在一个变量中。另外，它还将“做”和“测试”这两个步骤分开了。我个人觉得，这样才做到了条理清晰、重点突出。通常，假如我在一个if语句的条件部分调用了一个方法，那个方法主要的作用就应该是返回一个布尔值。但在这里，返回值的重要性好像比不上输出参数。读代码时，很容易就会遗漏一个方法调用中传递的输出参数。这会给读代码的人带来困扰。他们会奇怪是什么在负责实际的工作，以及为什么能像变魔术一样给出答案。使用可空版本，一切都会更直观——方法的结果包含我们感兴趣的全部信息。我在许多地方都采用了这个技术（一般是在有很多方法参数的情况下，此时，输出参数变得更难发现），它确实有助于改善代码的可读性。当然，这只适合值类型[①]。

这个模式的另一个好处在于，它可以和空合并操作符配合使用。你可以尝试去判断几个输入，在遇到第一个有效的输入后停止。普通的TryXXX模式允许用短路操作符来做到这一点，但由于要在同一个语句中为两个不同的输出参数使用相同的变量，所以含义显得不是特别清楚。

说明　**或者使用元组**　使用可空结果的另一种方式是，使用一个能够返回两个不同成员的返回类型，一个成员表示成功或失败，另一个表示成功时的值。Nullable<T>是很方便的，因为它提供了一个Boolean属性和一个T类型的属性，但返回值的含义也许可以更明确一些。.NET 4包含很多Tuple类型，而这里Tuple<int, bool>要比int?整洁一些。如果用自定义类型来表示解析操作的结果会更加整洁，例如ParseResult<T>。在这种情况下，向其他代码传递该值时，不用担心会引起混淆，而且还可以添加额外的信息，如导致解析失败的原因。

① 为什么适合值类型呢？是因为null对于值类型来说是一个“非自然”的值，所以用null可以很明显地表明失败。——译者注

下一个模式可用于解决另一个难题——多级比较（multitiered comparison）时的痛苦。

4.4.2 空合并操作符让比较不再痛苦

谁都不愿意反复写这么多相同的代码。重构往往能摆脱重复，但在某些情况下，重构也会面临很大的阻力。根据我个人的经验，用于Equals和Compare的代码就属于这种情况。

假定现在要写一个电子商务网站，而且已经有一个产品列表。你希望按流行度对产品进行排序（降序），然后按价格，再按名称排序。这样一来，五星级的产品就可以排在最前面，在这些产品中，最便宜的又排在较贵的产品前面。假如多个产品有相同的价格，那么以A开头的产品就排在以B开头的产品的前面。这其实并不是电子商务网站特有的问题。按多个标准对数据进行排序是一种相当常见的计算要求。

假定已经有一个恰当的Product类型，那么在C# 1中，可以像下面这样写比较代码：

```
public int Compare(Product first, Product second)
{
   // Reverse comparison of popularity to sort descending
   int ret = second.Popularity.CompareTo(first.Popularity);
   if (ret != 0)
   {
      return ret;
   }
   ret = first.Price.CompareTo(second.Price);
   if (ret != 0)
   {
      return ret;
   }
   return first.Name.CompareTo(second.Name);
}
```

假定上述代码不对空引用进行比较，所有属性返回的也都是非空的引用。可以使用一些预先进行的空比较以及Comparer<T>.Default来处理这些情况，但那会使代码变得更长、更复杂。可以稍稍重新安排一下，使代码变得更短（并避免从方法的中部返回）。但是，基本的“比较，检查，比较，检查”模式没有变。另外，不太容易注意到的一点是，我们得到了一个非零的答案后，其实就可以“收工”了。

最后一句话是不是让你想起了什么？对了，就是空合并操作符。如4.3节所示，如果有大量表达式以??分隔，就会反复应用这个操作符，直到它遇到一个非空的表达式。现在，我们唯一需要实现的就是如何从一次比较中返回空而不是零。用一个单独的方法来做这些事情比较容易，而且也可以在这个方法中封装使用默认的比较器。如果愿意，甚至可以编写一个重载方法，使用特定的比较器。另外，我们还要处理在传递的两个Product引用中，有一个为空的情况。

首先看看实现了辅助方法的类，如代码清单4-6所示。

代码清单4-6 提供“部分比较”的辅助类

```
public static class PartialComparer
{
```

```
    public static int? Compare<T>(T first, T second)
    {
       return Compare(Comparer<T>.Default, first, second);
    }

    public static int? Compare<T>(IComparer<T> comparer,
                                  T first, T second)
    {
       int ret = comparer.Compare(first, second);
       return ret == 0 ? new int?() : ret;
    }

    public static int? ReferenceCompare<T>(T first, T second)
       where T : class
    {
        return first == second ? 0
             : first == null ? -1
             : second == null ? 1
             : new int?();
    }
}
```

代码清单4-6中的Compare方法简单得"可怜"，如果没有指定一个比较器，就使用类型默认的comparer，而且在比较之后，对返回值进行的唯一处理就是：如果返回值是零，就转换成null。

说明　空值和条件操作符　在第二个Compare方法中，我在返回空值时使用了new int?()，而不是null，你可能会对此感到诧异。但是条件操作符要求它的第二个和第三个操作数要么为相同的类型，要么存在隐式转换。因此这里不能使用null，因为编译器不知道这个值表示的是什么类型。语言规则在检查子表达式时，不会考虑语句（返回int?类型的方法的返回语句）的总体目标。其他方法还包括将操作数显式转换为int?或使用default(int?)表示空值。基本上，重要的就是要确保其中一个操作数为int?值。

ReferenceCompare方法使用了另一个条件操作符——实际上，是三个。你会发现这比等价的if/else语句块可读性要差（尽管前者更长），这取决于你对条件操作符的熟练程度。我喜欢这种用法，因为它使比较的顺序更加清晰。同样，对于包含两个object参数的非泛型方法亦可如此，这种形式可以防止你不小心通过装箱来比较两个值类型。该方法只能用于引用类型，这是由类型参数约束决定的。

虽然这个类很简单，但用处可不小。现在可以将前面的产品比较替换成一个更简洁的实现：

```
public int Compare(Product first, Product second)
{
   return PC.ReferenceCompare(first, second) ??
          // Reverse comparison of popularity to sort descending
          PC.Compare(second.Popularity, first.Popularity) ??
          PC.Compare(first.Price, second.Price) ??
          PC.Compare(first.Name, second.Name) ??
          0;
}
```

你或许已经注意到，我使用的是PC而不是PartialComparer，这只是为了使代码只占纸张的一行。在实际的源代码中，我会使用完整的类型名称，而且每行一个比较。当然，如果出于某种原因需要短的代码行，那么可以指定一个using指令，使PC成为PartialComparer的别名。不过我并不建议这样做。

最后的0指明假如前面的所有比较都通过，那么两个Product实例是相等的。也可以使用Comparer<string>.Default.Compare(first.Name, second.Name)作为最后一个比较，但那会破坏方法的“对称性”。

这个比较能很好地处理空值，易于修改，建立了一个很好的模式供其他比较使用，而且它不会进行多余的比较：假如价格不同，名称就不必比较了。

你或许会想，相同的技术是否能应用于具有相似模式的相等性测试。不过，这样做的意义并不大。这是因为在经过了可空性和引用相等性（reference equality）测试之后，可以直接使用&&为布尔值提供所需的短路求值功能。然而，利用返回一个bool?的方法，我们可以根据引用来获取一个初始的绝对相等（definitely equal）、绝对不等（definitely not equal）或者未知（unknown）结果。PartialComparer的完整代码可通过本书的网站来获取，其中包含了适用的工具方法和实际使用它的例子。

4.5　小结

面临一个问题时，开发者的习惯是采取最容易的“短期”方案，即使它并不是特别“优雅”。这通常都是正确的决策，毕竟，我们可不想背负“过度工程”（overengineering）[①]的罪名。然而，假如一个好的方案也是最容易的方案，那么当然是再好不过了。

可空类型解决的是一个非常具体的问题。在C# 2以前，这个问题只能用一些比较“难看”的方案来解决。C# 1的老方案虽然可行，但执行起来比较花时间。现在利用新的特性，可以获得更好的支持。将泛型（为避免代码重复）、CLR支持（提供合适的装箱和拆箱行为）以及语言支持（提供简练的语法、方便的转换和操作符）相结合，使现在的方案变得更引人注目。

在提供可空类型的过程中，C#和Framework的设计者还设计出其他一些模式。在以前，这些模式不值得我们花费大量力气去实现。本章已演示了其中的一些模式。随着时间的推移，相信还会出现更多的模式，我对此一点也不会感到惊讶。

到目前为止，我们已讨论了两个新特性：泛型和可空类型。在C# 1中，正是由于缺少这两个特性，所以有时不得不忍受难闻的代码坏味道。下一章将延续这种写作风格，我们将讨论对委托的增强。这些增强构成了C#语言和.NET Framework发展方向的微妙变化，它们正在朝一个更“函数化”的方向发展。在C# 3中，你可以更清楚地看到这个发展趋势，因此，当我们还没有完全看到这些特性时，C# 2对委托的增强在C# 1中为我们所熟知的特性与C# 3中颇具潜力的“革命性”的编程风格之间起到了一个桥梁的作用。

① overengineering的意思是让自己的设计过于复杂和可靠，最终可能会影响整体的工作效率。——译者注

第 5 章

进入快速通道的委托

本章内容

- 啰嗦的C# 1语法
- 简化的委托构建
- 协变性和逆变性
- 匿名方法
- 捕获的变量

委托在C#和.NET中的发展历程非常有趣，它展示了设计者非比寻常的预见力（或者说是好运气）。在.NET 1.0/1.1中，“用事件处理程序来解决问题”这个思路并没有引起开发者太大的共鸣——直至C# 2出现。同样，设计者在C# 2的“委托”上付出了的巨大的努力，但人们并不是特别“买账”——直到它们在符合C# 3语法规则的代码中广泛使用。换言之，我们感觉语言和平台的设计者对未来的发展方向有一个大致的把握，但要等到多年之后，具体的发展方向才会变得清晰起来。

当然，C# 3本身并不是语言发展历程上的“终点站”，C# 4中的泛型委托就更灵活了，C# 5中异步委托写起来很容易，将来可能会看到在委托方面更大的进步。但是，C# 1和C# 3在这方面的差异是巨大的。（就C# 3对委托的支持来说，主要的改变就是Lambda表达式，这将在第9章详述。）

C# 2的委托就好像铺路石。它的新特性为C# 3更激动人心的变革铺平了道路，在让开发者感到非常舒服的同时，还提供了很多有用的好处。语言设计者们意识到，将C# 2特性组合在一起可以让我们以一种全新的方式来审视代码，不过他们并不知道这是一种什么样的方式。截至目前，他们的直觉证明了对于委托来说，这种方式有着非常显著的好处。

在.NET 2.0中，委托扮演了比在早期版本中更突出的角色（虽然都不及在.NET 3.5中使用广泛）。第3章演示了如何利用委托将一种类型的列表转换成另一种类型的列表。另外，第1章曾用`Comparison`委托而不是`IComparer`接口对一个产品列表进行排序。虽然框架和C#似乎始终很有礼貌地保持着一定的距离，但我相信语言和平台在这个领域是相互促进的：包含越来越多基于委托的API调用开始支持C# 2增强的语法，反之亦然。

本章将展示C# 2如何仅仅通过两处小改动，就极大简化了从普通方法创建委托实例的过程。

然后，我们将探讨它最大的一处变化：匿名方法。匿名方法允许在一个委托实例的创建位置内联地指定其操作。本章篇幅最大的一节将着眼于匿名方法最复杂的一个部分，也就是“捕获变量”（captured variable）。这种变量为委托实例提供了一个更好的工作环境。考虑到捕获变量的重要性和复杂性，我们将详尽讨论这个主题。只要理解了匿名方法，Lambda表达式的学习就会变得相对简单。

不过，首先还是回忆一下C# 1的委托机制的不便之处。

5.1 向笨拙的委托语法说拜拜

C# 1的委托语法看起来似乎并不太坏——语言已围绕`Delegate.Combine`、`Delegate.Remove`以及委托实例的调用提供了语法糖。至于要在创建委托实例时指定委托类型也是有道理的。毕竟，其他类型的实例也是用这种语法来创建的。

表面上，一切都在正常的轨道上，但不知因为什么，就是感觉不太对。很难确切地描述C# 1的委托创建表达式为什么会令人不快，但它们确实如此——至少对我而言。在C# 1中，我们一般是先写好一连串事件处理程序，然后到处写`new EventHandler`（或者其他要求的东西）。这显得很多余、很凌乱，因为事件本身已经指定了它要使用哪个委托类型。当然，仁者见仁，你可能会争辩说，在阅读C# 1风格的事件处理程序代码时，不需要进行过多的猜测。但是，由于代码中的文字量过多，会妨碍我们的阅读，并会使我们分心而忽略了真正应该注意的代码：你想用哪种方法来处理事件。

考虑将协变性和逆变性应用于委托时，问题就变得更简单了。假定现在有一个事件处理方法，它用于保存当前文档，或者只是简单地记录一下方法“被调用了”，或者执行其他不需要知道事件细节的操作。事件本身不应介意你的方法只能处理由`EventHandler`签名提供的信息，即使它声明为传递鼠标事件细节。遗憾的是，在C# 1中，必须为不同的事件处理程序的签名提供不同的方法。

同样，不可否认很别扭的一点在于：有时，我们写的方法是如此简单，以至于它们的实现比签名都要短。而这一切只是由于委托需要以方法的形式来执行代码。这样一来，在创建委托实例的代码和调用委托实例时应该执行的代码之间，就“多绕了一道弯子”。这道多余的“弯子”有时是受欢迎的，而且理所当然地，C# 2并不阻止你继续那样做。但与此同时，它会使代码难以阅读，并在类中填充了大量只用于委托的方法，对类造成了“污染”。

不出所料，所有这些在C# 2中都得到了极大的改进。虽然语法偶尔仍显啰嗦（这正是C# 3的Lambda表达式要解决的问题），但产生的差异是明显的。为了证明以前的种种不便，先来看一些C# 1代码，并在接下来的两个小节中对其进行改进。代码清单5-1生成了一个非常简单的窗体，上面有一个按钮，并订阅了3个按钮事件。

代码清单5-1　订阅的3个按钮事件

```
static void LogPlainEvent(object sender, EventArgs e)
{
   Console.WriteLine("LogPlain");
```

```
}

static void LogKeyEvent(object sender, KeyPressEventArgs e)
{
   Console.WriteLine("LogKey");
}

static void LogMouseEvent(object sender, MouseEventArgs e)
{
   Console.WriteLine("LogMouse");
}
...
Button button = new Button();
button.Text = "Click me";
button.Click      += new EventHandler(LogPlainEvent);
button.KeyPress   += new KeyPressEventHandler(LogKeyEvent);
button.MouseClick += new MouseEventHandler(LogMouseEvent);

Form form = new Form();
form.AutoSize = true;
form.Controls.Add(button);
Application.Run(form);
```

之所以将负责输出的行放到3个事件处理方法中，是为了证明代码在正常工作：如果在选定按钮时按空格键，那么会看到Click和KeyPress事件都被触发；按回车键只会触发Click事件；用鼠标单击按钮，会触发Click和MouseClick事件。在后续的小节中，我们将利用C# 2的一些特性来改进这些代码。

首先要求编译器执行一个非常明显的推断——在订阅一个事件时，我们想使用哪种委托类型呢？先来看看编译器能否做出如此简单的推断。

5.2　方法组转换

在C# 1中，如果要创建一个委托实例，就必须同时指定委托类型和要执行的操作。如果你记得第2章将操作定义成要调用的方法以及（对于实例方法来说）调用方法的目标。

例如，在代码清单5-1中，需要创建一个KeyPressEventHandler时，会使用如下表达式：

```
new KeyPressEventHandler(LogKeyEvent)
```

作为一个独立的表达式使用时，它并不“难看”。即使在一个简单的事件订阅中使用，它也是能够接受的。但是，在作为某个较长表达式的一部分使用时，看起来就有点“难看”了。一个常见的例子是在启动一个新线程时：

```
Thread t = new Thread(new ThreadStart(MyMethod));
```

同往常一样，我们希望以尽量简单的方式启动一个新线程来执行MyMethod。为此，C# 2支持从方法组到一个兼容委托类型的隐式转换。方法组（method group）其实就是一个方法名，它可以选择添加一个目标——换言之，和在C# 1中创建委托实例使用的表达式完全相同。（事实上，表达式当时就已经叫做“方法组”，只是那时还不支持转换。）如果方法是泛型的，方法组也可以

指定类型实参，不过根据我的经验，很少会这么做。新的隐式转换允许我们将事件订阅转换成：

```
button.KeyPress += LogKeyEvent;
```

类似地，线程的创建代码可简化成：

```
Thread t = new Thread(MyMethod);
```

如果只看一行代码，原始版本和改进的版本在可读性上的差异似乎并不大。但在代码量很大时，它们对可读性的提升就非常明显了。为了弄清楚到底发生了什么，我们简单看看这个转换具体都做了什么。

首先研究一下例子中出现的表达式LogKeyEvent和MyMethod。它们之所以被划分为方法组，是因为由于重载，可能不止一个方法适用。隐式转换会将一个方法组转换为具有兼容签名的任意委托类型。所以，假定有以下两个方法签名：


```
void MyMethod()
void MyMethod(object sender, EventArgs e)
```

那么在向一个ThreadStart或者一个EventHandler赋值时，都可以将MyMethod作为方法组使用：

```
ThreadStart x = MyMethod;
EventHandler y = MyMethod;
```

然而，对于本身已重载成可以获取一个ThreadStart或者一个EventHandler的方法，就不能把它（MyMethod）作为方法的参数使用——编译器会报告该转换具有歧义。同样，不能利用隐式方法组转换来转换成普通的System.Delegate类型，因为编译器不知道具体创建哪个委托类型的实例。这确实有点不方便，但使用显式转换，仍然可以写得比在C# 1中简短一些。例如：

```
Delegate invalid = SomeMethod;
Delegate valid = (ThreadStart)SomeMethod;
```

对于局部变量来说，这通常不是问题。但如果使用的API包含Delegate类型的参数（如Control.Invoke），还是有点烦人。解决方案如下：使用辅助方法、强制转换或使用中间变量。下面是一个使用了MethodInvoker委托类型的示例，不包含任何参数，也没有返回值：

```
static void SimpleInvoke(Control control,
                         MethodInvoker invoker)
{
   control.Invoke(invoker);
}
...
SimpleInvoke(form, UpdateUI);                    ← 使用辅助方法
form.Invoke((MethodInvoker)UpdateUI);            ← 使用强制转换
MethodInvoker invoker = UpdateUI;                ┐ 使用局部变量
form.Invoke(invoker);                            ┘
```

针对不同的情况可以使用不同的方案。尽管这些方案都不十分给力，但也不是很糟糕[①]。

和泛型一样，确切的转换规则要比这里讲的稍微复杂一些，我同样要鼓励你多去尝试。如果

① 在使用C# 3时，扩展方法（参见第10章）可以让辅助方法这个方案更加优雅。

编译器抱怨没有足够的信息,那么只需告诉它要使用什么转换就可以了。如果它没有抱怨(报错),就证明你的路子走对了。欲知详情，请参考语言规范的6.6节。可以进行的转换也许会比你预想的多，下一节将介绍。

5.3　协变性和逆变性

我们已在不同的上下文背景中对协变性和逆变性的概念进行了大量讨论。一般情况下，我们都是在叹息它们在C#中的“缺失”。但是，在C# 4之前版本的“委托构建”这一领域，它们是真实存在而且可用的。如果想比较细致地重温这两个术语的含义，请回过头去参考2.2.2节。如果不愿意翻到前面去，这里可以简单总结一下它们和委托联系起来时的要点：在静态类型的情况下，如果能调用一个方法，而且在能调用一个特定委托类型的实例并使用其返回值的任何地方都能使用该方法的返回值，就可以用该方法来创建该委托类型的一个实例。这听起来像绕口令，举一个例子就很容易明白了。

说明　不同的版本，不同的可变性类型　你可能已经意识到C# 4为委托和接口提供了泛型协变和逆变。不过这与我们此处所说的可变性完全不同。我们此刻只处理创建委托的新实例。C# 4中的泛型可变性使用引用转换，它不会创建新的对象，只是将已有的对象看成是不同的类型。

我们先来看一个逆变的例子，然后是协变。

5.3.1　委托参数的逆变性

现在考虑一下代码清单5-1中Windows Forms小应用中的事件处理程序。涉及的3个委托类型的签名[①]如下：

```
void EventHandler(object sender, EventArgs e)
void KeyPressEventHandler(object sender, KeyPressEventArgs e)
void MouseEventHandler(object sender, MouseEventArgs e)
```

注意，`KeyPressEventArgs`和`MouseEventArgs`都从`EventArgs`派生（其他许多类型也都是这样的——在本书写作的时候，MSDN上列出了从.NET 4中的`EventArgs`直接派生的403个类型）。所以，如果有方法要获取一个`EventArgs`参数，那么始终都可以在调用它时改为传递一个`KeyPressEventArgs`实参。所以，用签名与`EventHandler`相同的方法来创建`KeyPressEventHandler`的实例是完全合乎情理的，而那正是C# 2所做的。这是参数类型逆变性的一个例子。

为了体验一下实际的例子，让我们回头看看代码清单5-1，并假定我们不需要知道具体触发的是什么事件——我们只想指出“事件已发生”这一事实。利用方法组转换和逆变性，我们的代码会变得简单得多，如代码清单5-2所示。

---

① 考虑到一行可能写不下，我删除了每个签名最开头的`public delegate`部分。

代码清单5-2 演示方法组转换和委托逆变性

```
static void LogPlainEvent(object sender, EventArgs e)      ❶ 处理所有事件
{
   Console.WriteLine("An event occurred");
}
...
Button button = new Button();
button.Text = "Click me";                    ❷ 使用方法组转换
button.Click += LogPlainEvent;
button.KeyPress += LogPlainEvent;                 ❸ 使用转换和逆变性
button.MouseClick += LogPlainEvent;

Form form = new Form();
form.AutoSize = true;
form.Controls.Add(button);
Application.Run(form);
```


在这个程序中，我们彻底移除了专门处理按键和鼠标事件的两个事件处理方法，取而代之的是用一个事件处理方法❶来处理所有事件。当然，假如你希望为不同类型的事件执行不同的操作，这样做是起不到什么作用的。但有时，我们只想知道已经发生了一个事件，并可能想知道事件的来源。对Click事件的订阅❷只使用了上一节讨论的隐式转换，因为它有一个简单的EventArgs参数。但是，其他事件订阅❸由于要获取不同的参数类型，所以会涉及转换和逆变性。

前面讲过，.NET 1.0/1.1事件处理方法的约定在最初问世时并没有太大意义。这个例子准确地证明了当初的设计为什么在C# 2中才变得真正有用。根据约定，事件处理方法的签名应包含两个参数。第1个参数是object类型，代表事件的来源；第2个参数则负责携带与事件有关的任何额外信息，它的类型派生自EventArgs。在提供对逆变性的支持以前，这个设计没有太大的用处——让提供信息的参数从EventArgs派生并没有什么好处，而且有的时候，事件的来源也没有太大用处。通常，更合理的做法是直接以一个普通参数的形式来传递相关的信息，就像其他方法那样。但现在，你可以使用一个具有EventHandler签名的方法，作为符合约定的所有委托类型的操作。

我们已经介绍了传入方法或委托中的值，那么传出的值是什么样的呢？

5.3.2 委托返回类型的协变性

演示返回类型的协变性就要稍难一些，因为.NET 2.0中只有相当少的内建委托将返回类型声明为非void，并且它们还将返回值类型。但更容易的办法是声明我们自己的委托类型，并使用Stream作为返回类型。为简化讨论，我们将其设计成无参数的①：

```
delegate Stream StreamFactory();
```

现在只要声明一个方法返回一个特定的流类型，它就可以和这个委托一起使用，如代码清单5-3所示。在这个程序中，声明一个总是返回含有一些连续数据（字节0，1，2…直到15）的MemoryStream方法，并将该方法作为一个StreamFactory委托实例的操作来使用。

① 返回类型协变性和参数类型逆变性可以同时使用，虽然这样做几乎没有任何实际的用处。

代码清单5-3　演示委托返回类型的协变性

```
delegate Stream StreamFactory();                      ❶ 声明返回Stream的委托类型

static MemoryStream GenerateSampleData()              ❷ 声明返回MemoryStream的方法
{
   byte[] buffer = new byte[16];
   for (int i = 0; i < buffer.Length; i++)
   {
      buffer[i] = (byte) i;
   }
   return new MemoryStream(buffer);
}
...                                                   ❸ 利用协变性来转换方法组
StreamFactory factory = GenerateSampleData;

using (Stream stream = factory())                     ❹ 调用委托以获得Stream
{
   int data;
   while ((data = stream.ReadByte()) != -1)
   {
      Console.WriteLine(data);
   }
}
```

在代码清单5-3中，实际生成和显示的数据只是让代码有点儿事情做。在这个程序中，重要的是加了标注的那几行。我们声明委托类型的返回类型是Stream❶，但GenerateRandomData方法❷的返回类型是MemoryStream。负责创建委托实例的那一行❸执行前面提到的转换，并利用返回类型的协变性来允许GenerateSampleData用于StreamFactory。到调用委托实例时❹，编译器已经不知道返回的是一个MemoryStream——如果将stream变量的类型变成MemoryStream，会报告一个编译错误。

利用协变性和逆变性，还可基于一个委托实例来构造另一个委托实例。例如以下两行代码（假设已经有一个合适的HandleEvent方法）：

```
EventHandler general = new EventHandler(HandleEvent);
KeyPressEventHandler key = new KeyPressEventHandler(general);
```

第一行在C# 1中是有效的，第二行则不然——在C# 1中，要基于一个委托构造另一个，所涉及的两个委托类型的签名必须匹配。例如，你可以基于一个ThreadStart来创建一个MethodInvoker，但不能像上面的第二行代码那样，通过EventHandler创建一个KeyPressEventHandler。在上面的代码中，我们是在逆变性的帮助下，基于一个现有的委托实例来创建一个新的委托实例，这个现有的委托实例具有一个兼容的委托类型签名。在C# 2中，对“兼容”的定义要比C# 1宽松。

所有这些都很好，但美中不足的是还存在一个小瑕疵。

5.3.3　不兼容的风险

C# 2的这种新的灵活性会使本来有效的C# 1代码在C# 2编译时产生不同的结果。假设一个派

生类重载了某个基类中声明的方法，我们打算使用方法组转换创建一个委托的实例。由于C# 2中的协变性和逆变性，一个以前只和基类方法匹配的转换，现在也和派生类方法相匹配。在这种情况下，编译器将选择派生类方法。代码清单5-4演示了这种情况。

代码清单5-4　演示C# 1和C# 2之间的一处重大改变

```
delegate void SampleDelegate(string x);

public void CandidateAction(string x)
{
   Console.WriteLine("Snippet.CandidateAction");
}

public class Derived : Snippet
{
   public void CandidateAction(object o)
   {
      Console.WriteLine("Derived.CandidateAction");
   }
}
...
Derived x = new Derived();
SampleDelegate factory = new SampleDelegate(x.CandidateAction);
factory("test");
```

记住，Snippy①将在一个名为Snippet的类中生成所有这些代码，嵌套的类从这个类派生。在C# 1中，代码清单5-4会打印Snippet.CandidateAction，因为获取object参数的那个方法与SampleDelegate不兼容。但在C# 2中，它是兼容的。另外，由于是在一个派生的类型中声明的，所以选中的是这个方法，最终打印的将是Derived.CandidateAction。

幸好，C# 2编译器知道这是一处重大的改变，所以会发出相应的警告。我之所以写下这部分内容，是让你意识到有可能存在此类问题，但我确信这在现实生活中非常罕见。

可能的坏处已经说得够多了。接下来讨论和委托有关的最重要的一个新特性：匿名方法。它要比以前讨论过的主题稍微复杂一些，但它同时也是一个非常强大的特性——迈向C# 3的一大步。

5.4　使用匿名方法的内联委托操作

在C# 1中，用特定的签名来实现委托是很常见的情况，即使已经存在某个方法，它包含正确的行为但参数集稍有不同。同样，你只需要一个委托，做一件非常小非常小的事情，但也必须创建一个完整的新方法。该方法表示的行为只和原始方法有关，但现在却对整个类公开，这就在智能感知中产生了噪声，干扰了其功能。

这一切都让人极其沮丧。我们刚才提到的协变和逆变特性有时可以解决第一个问题，但通常都无能为力。同样是C# 2中新引入的*匿名方法*，则总是可以很漂亮地解决这些问题。

按照不太正式的说法，匿名方法允许你指定一个内联委托实例的操作，作为创建委托实例表

① 如果你跳过了本书的第1章，那么我告诉你Snippy是我用来创建短而完整的代码示例的一个工具。详情参见1.8.1节。

达式的一部分。匿名方法还以闭包（closure）的形式提供了一些更加强大的行为，但这方面的主题要到5.5节才会接触到。目前，还是先来看一些相对简单的主题。

首先，看一个能获取参数但不返回任何值的匿名方法。然后，分析提供返回值的语法，以及在不需要使用传递给我们的参数时，如何对程序进行简化。

5.4.1　从简单的开始：处理一个参数

.NET 2.0引入了一个泛型委托类型`Action<T>`，我们将在例子中使用该委托。它的签名非常简单（除了它是泛型这一事实以外）：

```
public delegate void Action<T>(T obj)
```

换言之，`Action<T>`就是对`T`的一个实例执行某些操作。所以，`Action<string>`可以反转字符串并打印出来，`Action<int>`可以打印传给它的那个数的平方根，而一个`Action< IList <double>>`可以计算出传给它的所有数的平均值并打印。纯属巧合，在代码清单5-5中，这些例子全部是用匿名方法来实现的。

代码清单5-5　将匿名方法用于`Action<T>`委托类型

```
Action<string> printReverse = delegate(string text)     ❶ 使用匿名方法创建 Action<string>
{
   char[] chars = text.ToCharArray();
   Array.Reverse(chars);
   Console.WriteLine(new string(chars));
};

Action<int> printRoot = delegate(int number)
{
   Console.WriteLine(Math.Sqrt(number));
};
Action<IList<double>> printMean = delegate(IList<double> numbers)
{
   double total = 0;
   foreach (double value in numbers)     ❷ 在匿名方法中使用循环
   {
      total += value;
   }
   Console.WriteLine(total / numbers.Count);
};

printReverse("Hello world");     ❸ 和调用普通方法一样调用委托
printRoot(2);
printMean(new double[] { 1.5, 2.5, 3, 4.5 });
```

代码清单5-5展示了匿名方法的几个不同的特性。首先是匿名方法的语法：先是`delegate`关键字，再是参数（如果有的话），随后是一个代码块，定义了对委托实例的操作。字符串反转代码❶表明可以在块中包含局部变量声明，而“列表求均值”代码❷演示了块中的循环。基本上，在普通方法体中能做的事情，在匿名方法中都能做。同样，匿名方法的结果是一个委托实例，可以像使用其他委托实例那样使用它❸。但要提醒你注意的是，逆变性不适用于匿名方法：必须指

定和委托类型完全匹配的参数类型。

> **说明** **一些限制** 有点奇怪的是，在值类型中编写匿名方法时，不能在其内部引用this。而在引用类型中则没有这个限制。此外，在微软C# 2和C# 3的编译器实现中，在匿名方法中通过base关键字访问基成员将导致警告——生成了无法验证的代码。C# 4编译器修复了这个问题。

说到实现，我们在IL中为源代码中的每个匿名方法都创建了一个方法：这时编译器将在已知类（匿名方法所在的类）的内部生成一个方法，并使用创建委托实例时的操作，就像它是一个普通方法一样[①]。CLR既不知道也不关心你使用了一个匿名方法。可以利用ildasm或者Reflector在已编译的代码中查看这些额外的方法。（Reflector知道怎样解释IL，在使用匿名方法的方法中显示那些匿名方法。但是，额外的方法仍然可见。）这些方法的名称是不友好（unspeakable）的，它们在IL中有效，但在C#中则无效。因此你无法在C#代码中直接引用它们，避免了可能的命名冲突。很多C# 2及以后版本的特性都用类似的方式实现。要辨识它们十分容易，因为它们通常都包含尖括号。例如，Main方法中的匿名方法可能会导致创建一个名为<Main>b__0的方法。不过这完全跟实现有关。比如微软很有可能在未来版本中更改私有约定。但这不会破坏任何东西，因为没有什么依赖于这些名称。

眼下有必要提醒你注意的是，和真实代码中的匿名方法相比，代码清单5-5展示的匿名方法稍显“臃肿”。在真实的代码中，它们常常作为传给另一个方法的参数使用（而不是赋给一个委托类型的变量），而且长度只有区区几行——毕竟，使用匿名方法的部分原因就是为了对代码进行精简。例如，第3章曾经指出，List<T>有一个ForEach方法，它获取一个Action<T>作为参数， ForEach将对每个元素执行该操作。代码清单5-6展示了这样的一个极端的例子，它用一种精简的形式实现代码清单5-5中的“求平方根”操作。

代码清单5-6 代码精简的极端例子。警告：可读性极差

```
List<int> x = new List<int>();
x.Add(5);
x.Add(10);
x.Add(15);
x.Add(20);
x.Add(25);

x.ForEach(delegate(int n){Console.WriteLine(Math.Sqrt(n));});
```

这非常可怕——乍一看，最后6个字符感觉就像是随机组合到一起的。如果既想精简，又想保证可读性，应该怎么办呢？事实上，确有折中的办法。我的习惯是在使用匿名方法时，就不再坚持“大括号单独占一行”这一规则（对于没有任何操作是属性，我也会这样），但仍然要保留足够的空白。所以，代码清单5-6的最后一行我通常会写成：

① 尽管总是会创建一个新的方法，但其位置却很难预料。5.5.4节将介绍这一点。

```
x.ForEach(delegate(int n)
   { Console.WriteLine(Math.Sqrt(n)); }
);

x.ForEach(delegate(int n) {
   Console.WriteLine(Math.Sqrt(n));
});
```

当然，仅向代码清单5-6中添加空白也是有帮助的。在所有的格式中，圆括号和大括号就不太容易让人犯晕了，人们很容易看明白代码在“做什么”。当然，如何分隔代码完全是你自己的事情，但我强烈建议你主动思考，以在简洁性和可读性之间取得平衡，并和其他团队成员商议，尽量在编码风格上取得一致。但是，一致的编码风格并非总是能够产生最易读的代码——有时，将所有内容都放到一行之内，反而更容易看懂。

到目前为止，我们只是通过参数和调用代码进行互动。那返回值呢？

5.4.2　匿名方法的返回值

`Action<T>`委托的返回类型是`void`，所以不必从匿名方法返回任何东西。为了演示在需要返回值时怎么办，将使用.NET 2.0中的`Predicate<T>`委托类型。下面列出了它的签名：

```
public delegate bool Predicate<T>(T obj)
```

代码清单5-7展示了一个匿名方法，它创建一个`Predicate<T>`的实例，其返回值指出传入的实参是奇数还是偶数。谓词（predicate）通常用于过滤和匹配——例如，可以利用代码清单5-7的代码来过滤一个列表，使之只包含偶数元素。

代码清单5-7　从匿名方法返回一个值

```
Predicate<int> isEven = delegate(int x) { return x % 2 == 0; };

Console.WriteLine(isEven(1));
Console.WriteLine(isEven(4));
```

新的语法和我们所期望的几乎完全相符——我们想把匿名方法当做一个普通的方法对待，并返回一个恰当的值。你可能以为还要在靠近`delegate`关键字的地方声明一个返回类型，但那是没有必要的。编译器只需检查是否所有可能的返回值都兼容于委托类型（编译器会尝试将匿名方法转换成这个委托类型）声明的返回类型[①]。

> **说明**　**从什么返回？**从匿名方法返回一个值时，它真的是只从匿名方法返回，不是从创建委托实例的方法返回。你可能会认为代码太容易，就不向下看，看到`return`关键字后，就草率地以为它是当前方法的一个退出点。

我在前面说过，.NET 2.0中很少有委托有返回值，但在第三部分将看到，.NET 3.5广泛使用了这一想法，特别是在使用LINQ时。`Comparison<T>`是另一个在.NET 2.0中常见的委托类型，

① `Predicate<T>`类型声明的返回类型恰好是`bool`。——译者注

可用来对集合排序。它是IComparer<T>接口的委托版。通常，一种情况下只需一个特定的排序顺序，所以采取内联的方式指定顺序是完全合理的，不需要在类的内部添加一个独立的方法来指定该顺序。代码清单5-8对此进行了演示，它输出C:\根目录下的文件名，并先按名称排序，再（分别）按大小排序。

代码清单5-8 用匿名方法简便地排序文件

```
static void SortAndShowFiles(string title, Comparison<FileInfo> sortOrder)
{
   FileInfo[] files = newDirectoryInfo(@"C:\").GetFiles();

   Array.Sort(files, sortOrder);
   Console.WriteLine(title);
   foreach (FileInfo file in files)
   {
      Console.WriteLine (" {0} ({1} bytes)", file.Name, file.Length);
   }
}
...
SortAndShowFiles("Sorted by name:", delegate(FileInfo f1, FileInfo f2)
   { return f1.Name.CompareTo(f2.Name); }
);

SortAndShowFiles("Sorted by length:", delegate(FileInfo f1, FileInfo f2)
   { return f1.Length.CompareTo(f2.Length); }
);
```

如果不使用匿名方法，就必须为每一种排序顺序都单独写一个方法。但在代码清单5-8中，每次调用SortAndShowFiles时，都可以清楚地看出要采用什么排序顺序。（有时，可以直接在调用匿名方法的位置调用Sort。但在代码清单5-8中，由于要分两次执行同一个fetch/sort/display操作序列，只是每次都选择不同的排序顺序，所以我将这些步骤封装到了它自己的方法中。）

有一种特殊的快捷语法可供利用。如果不关心委托的参数，那么根本不必声明它们。下面来看看它的工作原理。

5.4.3 忽略委托参数

在少数情况下，你实现的委托可能不依赖于它的参数值。你可能想写一个事件处理程序，它的行为只适用于一个事件，而不依赖于事件的实际参数值——比如保存用户的工作。事实上，在代码清单5-1描述的那个例子中，事件处理程序就完全符合这个标准。在这个例子中，可以完全省略参数列表，只需使用一个delegate关键字，后跟作为方法的操作而使用的代码块。代码清单5-9和代码清单5-1是完全等价的，只是它使用了更简洁的语法。

代码清单5-9 使用忽略了参数的匿名方法来订阅事件

```
Button button = new Button();
button.Text = "Click me";
button.Click      += delegate { Console.WriteLine("LogPlain"); };
button.KeyPress   += delegate { Console.WriteLine("LogKey"); };
```

```
button.MouseClick += delegate { Console.WriteLine("LogMouse"); };

Form form = new Form();
form.AutoSize = true;
form.Controls.Add(button);
Application.Run(form);
```

一般情况下，我们必须像下面这样写每一个订阅：

```
button.Click += delegate(object sender, EventArgs e) { ... };
```

那样会无谓地浪费掉大量空间——因为我们根本不需要参数的值，所以编译器现在允许完全省略参数。

我在实现自己的事件时，发现这种快捷语法相当好用。我已厌倦了每次在引发事件之前都要检查是否为空。为了消除这个烦恼，一个办法是确保事件一开始就有一个事件处理程序。并且永远都不会被删除。如果处理程序不做任何事情，你唯一损失的只有一点点性能。在C# 2之前，必须显式地创建一个具有恰当签名的方法，但这样做并非总是值得的。但现在，可以像下面这样做：

```
public event EventHandler Click = delegate {};
```

这样一来，以后就可以直接调用`Click`，无须检查是否有任何处理程序订阅了该事件。

但是，注意这个"参数通配"（parameter wildcarding）特性也存在一个陷阱：如果匿名方法能转换成多个委托类型（例如，为了调用不同的重载方法），那么编译器就需要你提供更多的辅助信息。为了对此进行演示，为此，我采用了一个比较麻烦的示例（演示方法组转换时使用的示例）：启动新线程。.NET 2.0有4个线程构造函数：

```
public Thread(ParameterizedThreadStart start)
public Thread(ThreadStart start)
public Thread(ParameterizedThreadStart start, int maxStackSize)
public Thread(ThreadStart start, int maxStackSize)
```

涉及的两个委托类型是：

```
public delegate void ThreadStart()
public delegate void ParameterizedThreadStart(object obj)
```

下面是创建一个新线程的3次尝试：

```
new Thread(delegate()          { Console.WriteLine("t1"); } );
new Thread(delegate(object o)  { Console.WriteLine("t2"); } );
new Thread(delegate            { Console.WriteLine("t3"); } );
```

第1行和第2行包含参数列表——编译器知道它不能将第1行中的匿名方法转换成一个`ParameterizedThreadStart`，或者将第2行中的匿名方法转换成一个`ThreadStart`。这2行都能成功编译，因为在每种情况下，都只有一个适用的构造函数重载版本。第3行则会产生歧义——匿名方法可以转换成两种委托类型，所以只获取一个参数的两个构造函数重载版本都是适用的。在这种情况下，编译器就无能为力了，它会报告一个错误。为了解决这个问题，一个办法是显式指定参数列表，另一个办法是将匿名方法强制转换为正确的委托类型。

希望迄今为止讲述的关于匿名方法的内容能激发你对自己的代码的一些思考，进而考虑在什么地方使用这些技术，从而取得好的效果。实际上，即使匿名方法的用处只有我们介绍过的这些，那它也非常有用，远非只是防止在你的代码中包含额外的方法那么简单。匿名方法是C# 2通过捕

获变量来实现的闭包。下一节将对这两个术语进行解释，并演示匿名方法的强大之处，以及一旦不小心会产生的混淆。

5.5　匿名方法中的捕获变量

我不喜欢迫不得已发出警告，但我感觉下面这个警告是必要的：如果你是初次接触这个主题，那么除非你感觉特别清醒，而且愿意花点时间来研究它，否则暂时不要开始本节的学习。但我又不希望给你没有必要的警告。事实上，只需花费一点精力，理解本节的主题并不太难。只是被捕获的变量会推翻你现有的一些知识和直觉，所以刚开始可能觉得它有点儿难。

不过，还是应该弄懂它。因为在掌握了本节的主题之后，代码的简洁性和可读性都会有很大的提高。另外，以后在研究C# 3中的Lambda表达式和LINQ时，这个主题也是至关重要的。所以，完全值得你花费精力去学习。

先从一些定义开始。

5.5.1　定义闭包和不同类型的变量

闭包是一个很古老的概念，最初是在Scheme中实现的。但是，随着近年来被越来越多的主流语言所接纳，人们对它的关注也越来越多。它的基本概念是：一个函数[①]除了能通过提供给它的参数交互之外，还能同环境进行更大程度的互动。但这个定义过于抽象，为了真正理解它在C# 2中的应用情况，还需理解另两个术语[②]。

- **外部变量**（outer variable）是指作用域（scope）内包括匿名方法的局部变量或参数（不包括`ref`和`out`参数）。在类的实例成员内部的匿名方法中，`this`引用也被认为是一个外部变量。
- **捕获的外部变量**（captured outer variable）通常简称为*捕获变量*（captured variable），它是在匿名方法内部使用的外部变量。重新研究一下“闭包”的定义，其中所说的“函数”是指匿名方法，而与之交互的“环境”是指由这个匿名方法捕获到的变量集。

所有这些解释听起来都“干巴巴”的，而且可能很难想象。但它主要强调的就是，匿名方法能使用在声明该匿名方法的方法内部定义的局部变量。这听起来似乎并不是一个了不起的设计，但在许多时候，它能带来巨大的便利——你可以使用现有的上下文信息，而不必专门设置额外的类型来存储你已经知道的数据。很快就会看到一些*有用的*、具体的例子，这一点我可以保证，但在此之前，有必要通过一些代码来明确上面那些定义。

在代码清单5-10的例子中包含大量局部变量。它是单独的一个方法，所以不能独自运行。这里不打算解释它的工作原理，或者它的功能，只是解释不同的变量是如何划分的。简化起见，我们再次使用了`MethodInvoker`委托类型。

① 这里的“函数”是常规意义上的计算机科学术语，而不是C#术语。

② 这两个术语定义在C# 4.0语言规范的7.14.4节。——译者注

代码清单5-10　不同种类的变量和匿名方法的关系

```
void EnclosingMethod()
{
   int outerVariable = 5;                          ❶ 外部变量（未捕获的变量）
   string capturedVariable = "captured";          ❷ 被匿名方法捕获的外部变量

   if (DateTime.Now.Hour == 23)
   {
       int normalLocalVariable = DateTime.Now.Minute;    ❸ 普通方法的局部变量
       Console.WriteLine(normalLocalVariable);
   }

   MethodInvoker x = delegate()
   {
       string anonLocal = "local to anonymous method";   ❹ 匿名方法的局部变量
       Console.WriteLine(capturedVariable + anonLocal);  ❺ 捕获外部变量
   };
   x();
}
```

我们由易到难来解释所有这些变量。

- ormalLocalVariable❸不是外部变量，因为它的作用域内没有匿名方法。它的行为和普通局部变量别无二致。
- anonLocal❹也不是外部变量，它是匿名方法的局部变量，但不是EnclosingMethod的局部变量。只有委托实例被调用之后，它才会存在[于一个正在执行的栈帧（frame）中]。
- outerVariable❶是一个外部变量，因为在它的作用域内声明了一个匿名方法。但是，匿名方法没有引用它，所以它未被捕捉。
- capturedVariable❷是一个外部变量，因为在它的作用域内声明了一个匿名方法，而且由于在❺这个位置使用了该变量，所以它成为了一个被捕捉的变量。

好了，虽然理解了术语，但我们对被捕捉的变量所做的事情并不是特别清楚。假设你能猜出运行代码清单5-10中方法后的输出结果。但在另一些情况下，结果可能会出乎你的预料。我们将从一个简单的例子开始，然后逐渐接触更复杂的。

5.5.2　捕获变量的行为

被匿名方法捕捉到的确实是变量，而不是创建委托实例时该变量的值。稍后就会看到它所产生的深远影响，但首先，我们必须理解对于相对简单的情况来说，这意味着什么。

代码清单5-11有一个捕获的变量和一个匿名方法，它们都能打印并改变变量。我们会看到，在匿名方法外部对变量的更改在匿名方法内部是可见的，反之亦然。

代码清单5-11　从匿名方法内外访问一个变量

```
string captured = "before x is created";

MethodInvoker x = delegate
{
   Console.WriteLine(captured);
```

```
    captured = "changed by x";
};

captured = "directly before x is invoked";
x();

Console.WriteLine(captured);

captured = "before second invocation";
x();
```

代码清单5-11的输出如下：

```
directly before x is invoked
changed by x
before second invocation
```

来看看具体是如何发生的。首先，声明变量captured，并将它的值设为一个十分普通的字符串字面量。到目前为止，变量没有任何特别的地方。然后，声明委托实例x，并将它的值设为捕获了captured的一个匿名方法。委托实例总是先打印captured的当前值，再把它更改为"changed by x"。需要注意的是，创建委托实例不会导致执行。

请记住，假如只是创建一个委托实例，不会读取变量，并将它的值存储到某个地方。为了证明这一点，现在将captured的值更改为"directly before x is invoked"。然后，我们第一次调用x。它会读取captured的值，并把它打印出来，从而产生第1行输出。然后，它会将captured的值设为"changed by x"并返回。委托实例返回后，这个“普通”的方法会照常进行，它会打印captured的当前值，从而产生第2行输出。

然后，普通方法[①]再次更改captured的值（这次修改为before second invocation），然后第二次调用x。captured的当前值会被打印出来，从而产生第3行输出。然后，委托实例将captured的值更改为"changed by x"并返回。此时，普通方法运行结束，整个程序结束。

虽然我们用这么多话解释了这么短一段代码的功能，但其中实际只有一个要点：*在整个方法中，我们使用的始终是同一个captured变量*。对一些人来说，这或许很难理解；对另一些人来说，这又或许是颇为自然。如果你觉得很难理解，不要担心，随着时间的推移，它必将变得越来越容易理解。

但是，即使你能很轻松地理解到目前为止我所讲的一切，也可能会产生这样的一个疑问：所有这一切到底有什么意义？OK，是时候展示一个真正有用的例子了。

5.5.3 捕获变量到底有什么用处

简单地说，捕获变量能简化避免专门创建一些类来存储一个委托需要处理的信息（除了作为参数传递的信息之外）。在ParameterizedThreadStart问世以前，如果你想启动一个新（非线程池）线程，并向其提供一些信息（比如要获取的一个网页的URL）就不得不创建一个额外的类型来容纳URL，并将ThreadStart委托实例的操作放到那个类型中。即使对于ParameterizedThreadStart来说，我们的方法也不得不接受一个object类型的参数，再将其强制转换为所需要的类型。这样一来，本来应该很简单的事情就被搞得很复杂。

① 也就是Main方法。——译者注

再来看看另一个例子，假定你有一个人物列表，并希望写一个方法来返回包含低于特定年龄的所有人的另一个列表。List<T>有一个FindAll方法能返回一个新列表，包含了和特定谓词匹配的所有内容。但是，在匿名方法和捕获变量问世之前，List<T>.FindAll[①]的存在并没有多大意义，因为要创建一个合适的委托，整个过程实在是太麻烦了。遍历整个列表并手动复制符合条件的项也许会简单一些。但是，在C# 2中，这一切变得非常容易：

```
List<Person> FindAllYoungerThan(List<Person> people, int limit)
{
   return people.FindAll(delegate (Person person)
      { return person.Age < limit; }
   );
}
```

我们在委托实例内部捕获了limit参数——如果仅有匿名方法而没有捕获变量，就只能在匿名方法中使用一个“硬编码”的限制年龄，而不能使用作为参数传递的limit。这无疑是一个非常巧妙的设计。和C# 1的版本相比，新的设计使我们能准确地描述自己的 “目的”，而不是将大量精力放在“过程”上。（在C# 3中，我们的代码甚至还能得到进一步的简化[②]。）你会很少遇到需要改写捕获变量的情况，但同样，这无疑也会有它的用处。

能跟上我的思路吗？很好。到目前为止，我们一直是在创建委托实例的那个方法内部使用委托实例。在这种情况下，你对捕获变量的生存期（lifetime）不会有太大的疑问。但是，假如委托实例“逃”到另一个黑暗的世界（big bad world），那么会发生什么？假如创建它的那个方法结束，它将何以应对？

5.5.4　捕获变量的延长生存期

在理解这种问题时，最简单的办法就是制定一个规则，给出一个例子，然后思考假如没有那个规则，会发生什么。下面就是规则，我们开始吧：

对于一个捕获变量，只要还有任何委托实例在引用它，它就会一直存在。

如果暂时没有头绪，请不要担心——例子就是帮助你理清头绪的。代码清单5-12展示了一个方法，它能返回一个委托实例。委托实例用捕获了一个外部变量的匿名方法来创建。那么，在方法返回之后，假如调用那个委托实例，会发生什么？

代码清单5-12　捕捉变量的生存期延长了

```
static MethodInvoker CreateDelegateInstance()
{
   int counter = 5;

   MethodInvoker ret = delegate
   {
      Console.WriteLine(counter);
      counter++;
```

① FindAll的参数是一个Predicate<T>委托。——译者注

② 如果你感到好奇，可以提前告诉你C# 3的写法是：return people.Where(person => person.Age < limit);。

```
    };
    ret();
    return ret;
}
...
MethodInvoker x = CreateDelegateInstance();
x();
x();
```

代码清单5-12的输出结果包括3行，每一行分别显示数字5、6和7。第1行输出是在CreateDelegateInstance内部调用委托实例的结果。这证明了counter的值在那个时候是可用的。但是，当方法返回之后呢？我们一般会认为counter在栈上，所以只要与CreateDelegate-Instance对应的栈帧被销毁，counter也会随之消失……但令人惊讶的是，以后调用返回的委托实例时，使用的似乎还是那个counter。

秘密在于前面那个假设，counter真的是在栈上吗？答案是否定的。事实上，编译器创建了一个额外的类来容纳变量。CreateDelegateInstance方法拥有对该类的一个实例的引用，所以它能使用counter。另外，委托也有对该实例的一个引用，这个实例和其他实例一样都在堆上。除非委托准备好被垃圾回收，否则那个实例是不会被回收的。

匿名方法的一些方面要严重依赖于编译器（不同的编译器可能使用不同的方式来支持相同的语义），但假如不用一个额外的类来容纳捕获变量，就很难明白指定的行为是如何实现的。要注意的是，如果只捕捉this，就不需要额外的类型了——编译器将直接创建一个实例方法来作为委托的操作。我之前提过，你不必过于担心栈和堆的细节，但有必要了解编译器能够实现什么样的东西，以防你对特定的行为是如何产生的感到困惑。

OK，现在你明白了，局部变量并非始终是“局部”的，即使在方法返回之后，它依然存在！你可能非常好奇我接下来会讲什么——现在，看看用多个委托来捕捉同一个变量的不同实例会发生什么？听起来是不是有些疯狂？这就是现在要讲述的一类问题。

5.5.5 局部变量实例化

运气不错时，捕获的变量的行为和我设想的基本一致，但运气差时情况很糟糕，由于我没有特别细心，结果令我很吃惊。如果出问题，几乎都是因为忘记了自己实际创建了多少个局部变量的“实例”。每当执行到声明一个局部变量的作用域时，就称该局部变量被*实例化*[①]。下面展示了一个简单的例子，它对两段非常相似的代码进行了比较：

```
int single;
for (int i = 0; i < 10; i++)
{
    single = 5;
    Console.WriteLine(single + i);
}
```

```
for (int i = 0; i < 10; i++)
{
    int multiple = 5;
    Console.WriteLine(multiple + i);
}
```

① 简单来说就是，每声明一次局部变量，它就被实例化一次。——译者注

在过去美好的日子里，这样的两段代码在语义上完全一致，所以通常会编译成相同的IL。假如不涉及任何匿名方法，现在仍将这样处理。局部变量所需的全部空间都在方法开始时在栈上分配，所以不会产生每次循环迭代都“重新声明”变量的开销①。但是，采用我们的新术语，single变量只实例化一次，而multiple变量将实例化10次——就像有10个局部变量，全部都叫做multiple，它们一个接一个地创建。

相信你现在已经明白我想说的话了——当一个变量被捕获时，捕捉的是变量的“实例”。如果在循环内捕捉multiple，第一次循环迭代时捕获的变量与第二次循环时捕获的变量是不同的，以此类推。代码清单5-13演示了由此产生的影响。

代码清单5-13　使用多个委托来捕捉多个变量实例

```
List<MethodInvoker> list = new List<MethodInvoker>();

for (int index = 0; index < 5; index++)
{
   int counter = index * 10;                      ❶ 实例化counter

   list.Add(delegate
   {                                              ❷ 打印并递增捕获的变量
      Console.WriteLine(counter);
      counter++;
   });
}

foreach (MethodInvoker t in list)
{                                                 ❸ 执行全部5个委托实例
    t();
}
list[0]();
list[0]();                                        ❹ 第1个委托多执行3次
list[0]();

list[1]();                                        ❺ 第2个委托多执行1次
```

代码清单5-13创建5个不同的委托实例❷——每次循环都创建一个。调用委托时，会先打印counter的值，再对它进行递增。现在，由于counter变量是在循环内部声明的，所以每次循环迭代，它都会被实例化❶。这样一来，每个委托捕捉到的都是一个不同的变量。所以，依次调用每个实例❸，就可以看到每次赋给counter的不同的初始值：0、10、20、30、40。为了加深你的理解，当返回第1个委托实例，再多执行它3次时❹，它会从那个实例的counter变量停止的地方继续，所以会输出1，2，3。最后，我们多执行一次第2个委托实例❺，这将从那个实例的counter变量停止的地方继续，所以会输出11。

所以，在这个例子中，每个委托实例都捕获了一个不同的变量。结束对这个例子的讨论之前，应该思考的一点是，假如捕捉的是index（由for循环声明的变量）而不是counter，那么会发生什么？在这种情况下，所有委托都将共享同一个变量。输出的将是数字5～13。之所以先输出5，是因为在循环终止之前，对index的最后一次赋值会把它设为5。不管涉及的是哪个委托，递增

① 在我看来，重新声明变量要更加整洁，除非需要在多个迭代之间维护这个值。

的都是同一个变量。foreach循环具有同样的行为：由循环的初始部分声明的变量只被实例化一次。这很容易弄错！如果你想捕获循环变量在一次特定的循环迭代中的值，必须在循环内部引入另一个变量，并将循环变量的值复制给它，再捕捉那个新变量——这正是我们在代码清单5-13中使用counter变量所做的事情。

> **说明　C# 5中的变化**　尽管for循环中的行为是合理的（毕竟变量只声明了一次），但在foreach中则有些意外。事实上，如果匿名方法超出了当前迭代，那么在其内部捕获迭代变量时通常都会产生错误。（如果委托实例仅用于迭代内部，则不会有问题。）这导致很多开发者深陷其中，以至于C#团队正在考虑在未来版本中改变foreach的语义，使其拥有更自然的行为——就好像每次迭代都拥有单独的变量。更多详细内容，请参见16.1节。

5

在最后一个例子中，来看一些非常糟糕的事情：只共享一部分捕获的变量，另一部分不共享。

5.5.6　共享和非共享的变量混合使用

在正式展示这个例子之前，我要明确的一点是：并不推荐你写这样的代码！事实上，之所以展示这个例子，我只是想向你证明：如果试图以过于复杂的方式来使用被捕捉的变量，情况很快就会变得很棘手。代码清单5-14创建了两个委托实例，每个都捕捉“相同”的两个变量。但是，仔细研究一下，就会发现实际发生的远比你想象得复杂。

代码清单5-14　捕捉不同作用域中的变量。警告：前面的代码非常糟糕

```
MethodInvoker[] delegates = new MethodInvoker[2];

int outside = 0;                                  ← ❶ 实例化变量一次

for (int i = 0; i < 2; i++)
{
   int inside = 0;                                ← ❷ 实例化变量多次

   delegates[i] = delegate
   {
       Console.WriteLine ("({0},{1})", outside, inside);
       outside++;                                    ❸ 使用匿名方法捕获变量
       inside++;
   };
}

MethodInvoker first = delegates[0];
MethodInvoker second = delegates[1];

first();
first();
first();

second();
second();
```

你花多少时间才能预测出代码清单5-14的输出（即使是在有注释的情况下）？老实说，它可花了我不少时间——比我愿意花在理解代码上的时间还要长。无论如何，把它作为一次练习，来看看具体发生了什么。

首先考虑一下outside变量❶。声明该变量的作用域只进入了一次[①]，所以很简单——它只有一个。inside变量❷则不同——每次循环迭代，都会实例化一个新的inside变量。这意味着当我们创建委托实例时❸，outside变量将由两个委托实例共享，但每个委托实例都有它们自己的inside变量。

循环结束后，我们创建的第1个委托实例被调用了3次。由于它每次都要对捕获到的变量进行递增，而且每个变量的初始值都是0，所以会看到先输出的是(0,0)，然后是(1,1)，再是(2,2)。执行第2个委托实例时，两个变量在作用域上的区别就变得非常明显了。在第2个委托实例中，有1个不同的inside变量，所以它的初始值仍为0，但共享的outside变量已经递增了3次。第2个委托实例被调用两次，所以，输出的先是(3,0)，然后是(4,1)。

出于兴趣，思考一下这是如何实现的——至少微软的C# 2编译器是如何实现的。这里发生的事情是：生成了一个额外的类，它包含外部变量（outside）；还生成了另一个额外的类，它包含内部变量（inside）和对第一个额外的类的引用。从根本上说，包含了一个捕获变量的每个作用域都有它自己的类型。在这个类型中，有一个引用指向下一个包含了捕获变量的作用域。在我们的例子中，类型的两个实例都包含着inside，这两个实例都含有同一个类型实例的引用，该类型实例中包含了outer类。其他的实现可能会采取别的做法，但这应该是最容易想到的一种做法。图5-1展示了代码清单5-14执行后的值。（这些名称可能并非编译器生成的确切名称，但是十分接近。注意，实际上委托实例可能还含有其他成员，但这里我们只关心目标。）

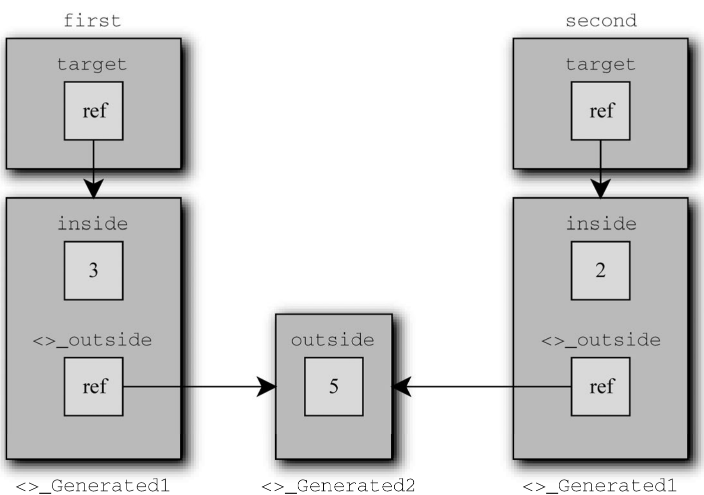


图5-1　内存中多个捕获变量作用域的简单说明

① 也就是说，它只声明了一次。——译者注

即使完全理解了代码，也可以继续把它作为模板使用，以便对捕获变量的其他方面进行试验。前面提到，对变量进行捕获时，某些方面是依赖于具体实现的[①]。通常，通过阅读语言规范以了解它所支持的功能是很有用的，但有时也需要多写写代码来看一看实际发生的事情。

虽然代码清单5-14的代码不好懂，但在某些情况下，为了表示一个你希望的行为，像那样的代码反而可能是最简单、最清楚的一种方式。但是，眼见为实。另外，那时，我也肯定会在代码中添加详尽的注释来解释发生的事情。那么，什么时候适合使用捕获变量？需要注意什么？

5.5.7 捕获变量的使用规则和小结

我希望通过本节的学习，你在使用捕获的变量时能特别小心。它们虽然在逻辑上讲得通（为使它们变得更简单而进行修改的企图，要么会使它们失去实用性，要么使它们失去逻辑性），但却很容易产生异常复杂的代码。

但是，不要因为这一点就害怕使用它们——它们能使你避免写大量枯燥的代码，而且假如运用得当，还能通过最容易让人理解的代码来完成工作。那么，怎样才算得当？

使用捕获变量时，请参照以下规则。

- 如果用或不用捕获变量时的代码同样简单，那就不要用。
- 捕获由`for`或`foreach`语句声明的变量之前，思考你的委托是否需要在循环迭代结束之后延续，以及是否想让它看到那个变量的后续值。如果不是，就在循环内另建一个变量，用来复制你想要的值。（在C# 5中，你不必担心`foreach`语句，但仍需小心`for`语句。）
- 如果创建多个委托实例（不管是在循环内，还是显式地创建），而且捕获了变量，思考一下是否希望它们捕捉同一个变量。
- 如果捕捉的变量不会发生改变（不管是在匿名方法中，还是在包围着匿名方法的外层方法主体中），就不需要有这么多担心。
- 如果你创建的委托实例永远不从方法中“逃脱”，换言之，它们永远不会存储到别的地方，不会返回，也不会用于启动线程——那么事情就会简单得多。
- 从垃圾回收的角度，思考任何捕获变量被延长的生存期。这方面的问题一般都不大，但假如捕获的对象会产生昂贵的内存开销，问题就会凸现出来。

第一条规则可奉为金科玉律。简化总是好事。所以在任何时候，如果使用一个捕获的变量能使代码变得更简单（前提是你已将强迫代码维护人员理解捕获的变量所做的事情这一额外复杂性考虑在内），那么就用它。但你也要考虑它所带来的额外的复杂度，不要一味地追求最少的代码量。

本节讨论了大量基础知识，我也意识到这些内容可能很难理解。我列出了一些要记住的重要知识点，以后需要温习本节的内容时，可以直接参考这些要点，不需要重新阅读所有内容。

- 捕获的是变量，而不是创建委托实例时它的值。
- 捕获的变量的生存期被延长了，至少和捕捉它的委托一样长。

---

① 换言之，不同的编译器可能有不同的实现方法。——译者注

- 多个委托可以捕获同一个变量……
- ……但在循环内部，同一个变量声明实际上会引用不同的变量“实例”。
- 在`for`循环的声明中创建的变量[①]仅在循环持续期间有效——不会在每次循环迭代时都实例化。这一情况对于C# 5之前的`foreach`语句也适用。
- 必要时创建额外的类型来保存捕获变量。
- 要小心！简单几乎总是比要小聪明好。

以后在讨论C# 3及其Lambda表达式时，会看到有更多的变量被捕捉。但就目前来说，我们已经完成了对新的C# 2委托特性的总结。（是不是感觉松了一口气？）

5.6　小结

C# 2根本性地改变了委托的创建方式，这样我们就能在.NET Framework的基础上采取一种更函数化的编程风格。与.NET 1.0/1.1相比，.NET 2.0中有更多以委托作为参数的方法。这一趋势在.NET 3.5中得到了延续。`List<T>`就是最好的例子，也是检验你使用匿名方法和捕获变量水平的一个很好的测试床。这种编程方式需要一种稍微不同的思维模式——你必须退一步想想，最终的目标是什么，它适合采用传统的C#方式来表示，还是适合采用一种更函数化的方式来表示。

对委托的处理方式进行的所有更改都是有用的，但它们确实增加了语言的复杂性，尤其是在遇到捕获的变量时。闭包在涉及如何对可用的环境进行共享时，实现起来总是有一定的难度。在这个问题上，C#并无例外。但是，闭包之所以能作为一种思想存在这么久，是因为它们确实能使代码变得更容易理解和更直接。在功能的复杂性和编程的简化性之间取得平衡，这始终都不是一件容易的事情，但这值得你去谨慎地尝试。久而久之，你应该能够更好地使用捕获变量并理解它们的行为。LINQ促进了它们的进一步使用，现如今，常见的C#代码中已经越来越多地使用了闭包。

在C# 2的新特性中，并非只有匿名方法才要求编译器在幕后创建额外的类型，并用“貌似”局部的变量悄悄地干一些不太光彩的事情。下一章会介绍更多这样的例子。届时你会看到，编译器实际是为我们构建了一个完整的状态机（state machine），使开发者能够更容易地实现迭代器。

---

① 比如代码清单5-13中的`index`变量。——译者注

第6章 实现迭代器的捷径

本章内容

- 在C# 1中实现迭代器
- C# 2中的迭代器块
- 迭代器使用示例
- 使用迭代器作为协同程序

迭代器模式是*行为模式*的一种范例，行为模式是一种简化对象之间通信的设计模式。这是一种非常易于理解和使用的模式。实际上，它允许你访问一个数据项序列中的所有元素，而无须关心序列是什么类型——数组、列表、链表或任何其他类型。它能非常有效地构建出一个*数据管道*，经过一系列不同的转换或过滤后再从管道的另一端出来。实际上，这也是LINQ的核心模式之一，第三部分将会介绍。

在.NET中，迭代器模式是通过`IEnumerator`和`IEnumerable`接口及它们的泛型等价物来封装的（命名上有些不恰当——涉及模式时通常称为*迭代*而非*枚举*，就是为了避免和枚举这个词的其他意思相混淆，本章将使用迭代器和可迭代的）。如果某个类型实现了`IEnumerable`接口，就意味着它可以被迭代访问。调用`GetEnumerator`方法将返回`IEnumerator`的实现，这就是迭代器本身。可以将迭代器想象成数据库的游标，即序列中的某个位置。迭代器只能在序列中向前移动，而且对于同一个序列可能同时存在多个迭代器操作。

作为一门语言，C# 1利用`foreach`语句实现了访问迭代器的内置支持。这让我们遍历集合时无比容易（比直接使用`for`循环要方便得多）并且看起来非常直观。`foreach`语句被编译后会调用`GetEnumerator`和`MoveNext`方法以及`Current`属性，假如`IDisposable`也实现了，程序最后还会自动销毁迭代器对象。这是一个虽不起眼但却很有用的语法糖。

然而，在C# 1中，实现迭代器是比较困难的。C# 2所提供的语法糖可以大大简化这个任务，所以有时候更应该去实现迭代器模式。否则会导致更多的工作量。

本章将研究实现迭代器所需的代码，以及C# 2所给予的支持。在详细介绍了语法之后，我们将研究一些现实世界中的示例，包括微软并发函数库（concurrency library）中对迭代器语法振奋人心的使用（虽然有点异乎寻常）。我会在描述完全部细节之后再提供示例，因为要学的内容并不多，而且在理解了代码的功能之后再去看示例，要清晰得多。如果要先看看示例，可以翻到6.3节和6.4节。

与其他章节一样，先看一下C# 2之前的版本是如何处理的。我们将用硬编码的方式来实现一个迭代器。

6.1　C# 1：手写迭代器的痛苦

之前研究对泛型集合进行迭代时，我们已经在3.4.3节中看到了一个实现迭代器的例子。在某些方面，它比使用C# 1实现迭代器要难，因为我们还实现了泛型接口，不过在某些方面，它又会容易一些，因为它实际上不需要迭代任何有用的东西。

为了能顺利过渡到C# 2的特性上，首先实现一个相对简单的迭代器，但它仍可以提供真实有用的值。假设我们有一个基于循环缓冲区的新的集合类型。我们将实现IEnumerable接口，以便新类的用户能轻松地迭代集合中的所有值。在这里，我们不关心这个集合中的内容是什么，仅仅关注迭代器的实现。这个集合将把值存储在一个数组中（就是object[]，这里没有使用泛型），并且集合有一个有趣的特性，就是能设置它的逻辑“起点”。所以如果数组有5个元素，并且你把起点设置为2，这样的话我们就看到元素2、3、4、0和1依次返回。这里不会展示完整的循环缓冲代码，你可以在可下载的代码中找到它们。

为了方便演示这个类，我们将在构造函数中设置值和起点。所以，编写代码清单6-1，来对集合进行迭代。

代码清单6-1　使用（还未实现的）新集合类型的代码

```
object[] values = {"a", "b", "c", "d", "e"};
IterationSample collection = new IterationSample(values, 3);
foreach (object x in collection)
{
   Console.WriteLine (x);
}
```

运行代码清单6-1应该（最终）会产生输出结果d、e、a、b和c，因为之前设置的起点为3。现在知道我们要完成的功能了，下面来看一下在代码清单6-2中这个类的框架。

代码清单6-2　新集合类型的框架，不包含迭代器的实现

```
using System;
using System.Collections;

public class IterationSample : IEnumerable
{
   object[] values;
   int startingPoint;

   public IterationSample(object[] values, int startingPoint)
   {
      this.values = values;
      this.startingPoint = startingPoint;
   }

   public IEnumerator GetEnumerator()
   {
```

```
            throw new NotImplementedException();
        }
    }
```

正如你看到的，现在还未实现GetEnumerator方法，不过其余的代码已经可以运行了。那么，要如何实现GetEnumerator方法呢？首先要知道，我们需要在某个地方存储某个状态。迭代器模式的一个重要方面就是，不用一次返回所有数据——调用代码一次只需获取一个元素。这意味着我们需要确定访问到了数组中的哪个位置。在了解C# 2编译器为我们所做的事情时，迭代器的这种状态特质十分重要，因此，要密切关注本例中的状态。

那么，这个状态值要保存在哪里呢？假设我们尝试把它放在IterationSample类自身里面，让它既实现IEnumerator接口又实现IEnumerable接口。乍一看，这似乎是个好主意——毕竟，我们是将数据保存在正确的位置，其中也包括了起点。GetEnumerator方法可以仅返回this。然而，使用这种方式存在一个大问题——如果GetEnumerator方法被调用了多次，那么就会返回多个独立的迭代器。例如，我们能使用两个嵌套的foreach语句，以便得到所有可能的成对值。这就意味着，两个迭代器需要彼此独立，每次调用GetEnumerator方法时都需要创建一个新对象。我们仍可以直接在IterationSample内部实现功能，不过只用一个类的话，分工就不明确——那样会让代码非常混乱。

因此，可以创建另外一个类来实现这个迭代器。我们将使用"C#嵌套类型可以访问它外层类型的私有成员"这一特点，就是说，我们仅需要存储一个指向"父级"IterationSample类型的引用和关于所访问到的位置的状态，如代码清单6-3所示。

代码清单6-3　嵌套类实现集合迭代器

```
class IterationSampleIterator : IEnumerator
{
    IterationSample parent;            ❶ 正在迭代的集合
    int position;                      ❷ 指出遍历到的位置

    internal IterationSampleIterator(IterationSample parent)
    {
        this.parent = parent;
        position = -1;                 ❸ 在第一个元素之前开始
    }

    public bool MoveNext()
    {
        if (position != parent.values.Length)   ❹ 如果仍要遍历，那么增加position的值
        {
            position++;
        }
        return position < parent.values.Length;
    }

    public object Current
    {
        get
        {
            if (position == -1 ||      ❺ 防止访问第一个元素之前和最后一个元素之后
```

```
                position == parent.values.Length)
            {
                throw new InvalidOperationException();
            }
            int index = position + parent.startingPoint;       ❻ 实现封装
            index = index % parent.values.Length;
            return parent.values[index];
        }
    }

    public void Reset()
    {                                                  ❼ 返回第一个元素之前
        position = -1;
    }
}
```

这么简单的一个任务竟然使用了这么多代码！我们要记住进行迭代的原始值的集合❶，并用简单的从零开始的数组跟踪我们所在的位置❷。为了返回元素，要根据开始点对索引进行偏移❻。为了和接口一致，要让迭代器逻辑上从第一个元素的位置之前开始❸，所以在第一次使用Current属性之前，调用代码必须调用MoveNext方法。❹中的条件增量可以保证❺中的条件判断简单准确，即使在程序第一次报告无可用数据后又调用MoveNext也没有问题。为了重置迭代器，我们将我们的逻辑位置设置回“第一个元素之前”❼。

这里涉及的大部分逻辑都非常简单，当然还是有大量的地方会出现“边界条件逻辑错误”（off-by-one error）。实际上，我的第一个实现就是因为这个原因没有通过单元测试。不过，幸好它现在可以正常运行了，现在只需在IterationSample中实现IEnumerable接口来完成这个例子：

```
public IEnumerator GetEnumerator()
{
    return new IterationSampleIterator(this);
}
```

这里没有复制合并在一起的代码，不过在本书的网站上有包括代码清单6-1在内的完整代码，它可以生成我们期望的输出。

要谨记这只是一个相对简单的例子——没有太多的状态需要跟踪，也没有尝试检查集合是否在两次迭代之中被改变。实现一个简单的迭代器都需要花费这么大的精力，所以很少有人能在C# 1中实现这个模式也不足为奇。开发人员通常喜欢用foreach在由框架提供的集合上执行迭代，或使用更直接（和集合特定）的方式来访问他们自己构建的集合。

因此，在C# 1中用了40行代码来实现迭代器。下面来看一下在C# 2中情况能否好转。

6.2　C# 2：利用 **yield** 语句简化迭代器

我是那种为了在圣诞节到来那一刻尽早打开礼物，宁愿在平安夜熬到凌晨不睡的人。同样，我（这种迫不及待的人）也不可能在向大家展示简洁的C# 2解决方法之前，浪费过多的时间。

6.2.1　迭代器块和**yield return**简介

如果C# 2不具备缩减实现迭代器所编写的代码量这个强大的特性，那么本章就没有存在的必要了。在其他主题中，代码量只会减少一点点，或者仅仅使编写出的代码更优雅。然而在这里，所需的代码量大大地减少了。代码清单6-4展示了在C# 2中GetEnumerator方法的完整实现。

代码清单6-4　利用C# 2和yield return来迭代示例集合

```
public IEnumerator GetEnumerator()
{
   for (int index = 0; index < values.Length; index++)
   {
      yield return values[(index + startingPoint) % values.Length];
   }
}
```

4行代码就搞定了，其中还有两行大括号。明确地讲，它们完全替换掉了整个IterationSampleIterator类。至少在源代码中是这样的……稍后，我们将看到编译器在后台都做了哪些工作，以及这个实现的奇特之处。不过此时还是先看一下这里用到的源代码。

在你看到yield return之前，这个方法看上去一直都非常正常。这句代码就是告诉C#编译器，这个方法不是一个普通的方法，而是实现一个迭代器块的方法。这个方法被声明为返回一个IEnumerator接口，所以就只能使用迭代器块来实现返回类型为IEnumerable、IEnumerator或泛型等价物的方法[①]。如果方法声明的返回类型是非泛型接口，那么迭代器块的生成类型（yield type）是object，否则就是泛型接口的类型参数。例如，如果方法声明为返回IEnumerable<string>，那么就会得到string类型的生成类型。

在迭代器块中不允许包含普通的return语句——只能是yield return。在代码块中，所有yield return语句都必须返回和代码块的生成类型兼容的值。在之前的例子中，不能在一个声明返回IEnumerable<string>的方法中编写yield return 1;这样的代码。

> **说明**　**对yield return的限制**　对yield语句有一些额外的限制。如果存在任何catch代码块，则不能在try代码块中使用yield return，并且在finally代码块中也不能使用yield return或yield break（这个语句马上就要讲到）。这并非意味着不能在迭代器内部使用try/catch或try/finally代码块，只是说使用它们时有一些限制而已。如果想了解为什么会存在这样的限制，可以查看Eric Lippert的一个博文系列，其中介绍了这种限制以及迭代器方面的其他设计决策，网址为http://mng.bz/EJ97。

编写迭代器块时，需要记住重要的一点：尽管你编写了一个似乎是顺序执行的方法，但实际上是请求编译器为你创建了一个状态机。编译器这样做的原因，和我们在C# 1的迭代器实现中塞

① 或者属性也可，我们后面会看到。但是不能在匿名方法中使用迭代器代码块。

入那么多代码的原因完全一样——调用者每次只想获取一个元素，所以在返回上一个值时需要跟踪当前的工作状态。

当编译器看到迭代器块时，会为状态机创建一个嵌套类型，来正确记录块中的位置以及局部变量（包括参数）的值。所创建的类类似于我们之前用普通方法实现的类，用实例变量来保存所有必要的状态。下面来看一下，要实现迭代器，这个状态机要做哪些事情：

- 它必须具有某个初始状态；
- 每次调用MoveNext时，在提供下一个值之前（换句话说，就是执行到yield return语句之前），它需要执行GetEnumerator方法中的代码；
- 使用Current属性时，它必须返回我们生成的上一个值；
- 它必须知道何时完成生成值的操作，以便MoveNext返回false。

要实现上述内容的第二点需要一定的技巧，因为它总是要从之前达到的位置“重新开始”执行代码。跟踪局部变量（当变量处于方法中时）不算太难——它们在状态机中由实例变量来表示。而重新启动[①]的动作更需技巧，不过好在你不用自己编写C#编译器，所以不用关心它是如何实现的，只要明白从黑盒出来的结果能正确工作就行。在迭代器块中，你可以编写普通代码，编译器负责确保执行流程和在其他方法中一样正确。不同的是，yield return语句只表示“暂时地”退出方法——事实上，你可把它当作暂停。

接下来，我们用更加形象的方式深入研究执行流程。

6.2.2　观察迭代器的工作流程

使用序列图的方式有助我们充分了解迭代器是如何执行的。我们不用手工绘制这个图，而是用程序把流程打印出来（代码清单6-5）。这个迭代器本身仅仅提供了一个数字序列（0,1,2,–1）。有意义的部分不是这些数字，而是代码的流程。

代码清单6-5　显示迭代器及其调用者之间的调用序列

```
static readonly string Padding = new string(' ', 30);

static IEnumerable<int> CreateEnumerable()
{
   Console.WriteLine("{0}Start of CreateEnumerable()", Padding);

   for (int i=0; i < 3; i++)
   {
      Console.WriteLine("{0}About to yield {1}", Padding, i);
      yield return i;
      Console.WriteLine("{0}After yield", Padding);
   }
   Console.WriteLine("{0}Yielding final value", Padding);
   yield return -1;

   Console.WriteLine("{0}End of CreateEnumerable()", Padding);
```

① 指继续执行yield return之后的代码。——译者注

```
}
...
IEnumerable<int> iterable = CreateEnumerable();
IEnumerator<int> iterator = iterable.GetEnumerator();
Console.WriteLine("Starting to iterate");
while (true)
{
   Console.WriteLine("Calling MoveNext()...");
   bool result = iterator.MoveNext();
   Console.WriteLine("... MoveNext result={0}", result);
   if (!result)
   {
      break;
   }
   Console.WriteLine("Fetching Current...");
   Console.WriteLine("... Current result={0}", iterator.Current);
}
```


代码清单6-5的代码确实不够优雅，尤其进行迭代的代码更是如此。在通常的写法中，我们一般使用foreach循环语句，不过为了完全地展现运行过程中何时发生何事，须把迭代器分割为一些很小的片段。这段代码的作用同foreach大致相同，然而foreach还会在最后调用Dispose方法，随后我们会看到，这对于迭代器块是很重要的。可以看到，虽然这次我们返回的是IEnumerable<int>而非IEnumerator<int>，但迭代器方法里的语法没有任何区别。通常为了实现IEnumerable<T>，我们只会返回IEnumerator<T>。如果你只想在方法中生成一个序列，可以返回IEnumerable<T>。

下面就是代码清单6-5输出的结果：

```
Starting to iterate
Calling MoveNext()...
                              Start of CreateEnumerable()
                              About to yield 0
... MoveNext result=True
Fetching Current...
... Current result=0
Calling MoveNext()...
                              After yield
                              About to yield 1
... MoveNext result=True
Fetching Current...
... Current result=1
Calling MoveNext()...
                              After yield
                              About to yield 2
... MoveNext result=True
Fetching Current...
... Current result=2
Calling MoveNext()...
                              After yield
                              Yielding final value
... MoveNext result=True
Fetching Current...
```

```
... Current result=-1
Calling MoveNext()...
                                    End of CreateEnumerable()
... MoveNext result=False
```

这个结果中有几个重要的事情需要牢记：

- 在第一次调用MoveNext之前，CreateEnumerable中的代码不会被调用；
- 所有工作在调用MoveNext时就完成了，获取Current的值不会执行任何代码；
- 在yield return的位置，代码就停止执行，在下一次调用MoveNext时又继续执行；
- 在一个方法中的不同地方可以编写多个yield return语句；
- 代码不会在最后的yield return处结束，而是通过返回false的MoveNext调用来结束方法的执行。

第一点尤为重要，因为它意味着如果在方法调用时需要立即执行代码，就不能使用迭代器块，如参数验证。如果你将普通检查放入用迭代器块实现的方法中，将不能很好地工作。你肯定会在某些时候违反这些约束——这是十分常见的错误，而且如果你不知道迭代器块的原理，也很难理解为什么会这样。6.3.3节将解决这个问题。

有两件事情我们之前尚未接触到——终止迭代过程的其他方式，以及finally代码块如何在这种有点古怪的执行形式中工作。现在来看一下。

6.2.3　进一步了解迭代器执行流程

在常规的方法中，return语句具有两个作用：第一，给调用者提供返回值；第二，终止方法的执行，在退出时执行合适的finally代码块。我们看到yield return语句临时退出了方法，直到再次调用MoveNext后又继续执行，我们根本没有检查finally代码块的行为。如何才能真正地停止方法？所有这些finally代码块发生了什么？我们从一个非常简单的构造（yield break语句）开始。

1. 使用yield break结束迭代器的执行

人们总希望找到某种方式来让方法具有单一的出口点，很多人也很努力地实现这个目标[①]。同样的技术也适用于迭代器块。不过，如果你希望“提早退出”，那么yield break语句正是你所需要的。它实际上终止了迭代器的运行，让当前对MoveNext的调用返回false。

代码清单6-6演示了在计数到100次的过程中，假如运行超过时限就提前停止的情况。它也演示了在迭代器块中使用方法参数[②]的方式，并证实这与方法的名称是无关的。

代码清单6-6　演示yield break语句

```
static IEnumerable<int> CountWithTimeLimit(DateTime limit)
{
   for (int i = 1; i <= 100; i++)
```

① 我个人认为，为实现这个目标你所使用的方法可能会使代码很难阅读，不如设置多个出口点，尤其在需要估计任何可能发生的异常并使用try/finally进行资源清理时。不过，关键是做到这一点不难。

② 记住，迭代器块不能实现具有ref或out参数的方法。

```
    {
        if (DateTime.Now >= limit)
        {
            yield break;          <—— 如果时间到了就停止运行
        }
        yield return i;
    }
}
...
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
    Console.WriteLine("Received {0}", i);
    Thread.Sleep(300);
}
```

正常情况下，运行代码清单6-6时将看到大约7行的输出结果。如我们所预计的那样，foreach循环能够正常地结束，迭代器遍历完了需要遍历的元素。yield break语句的行为非常类似于普通方法中的return语句。

到目前为止，一切都还相对简单。执行流程的最后一个要研究的地方是：finally代码块如何执行及何时执行。

2. finally代码块的执行

在要离开相关作用域时，我们习惯执行finally代码块。迭代器块行为方式和普通方法不太一样，尽管我们也看到，yield return语句暂时停止了方法，但并没有退出该方法。按照这样的逻辑，在这里我们不要期望任何finally代码块能够正确执行——实际也确实如此。不过，在遇到yield break语句时，适当的finally代码块还是能够执行的，正如在从普通方法中返回时你所期望的那样[①]。

finally在迭代器块中常用于释放资源，通常与using语句配合使用。6.3.2节将介绍一个真实的例子，但现在我们先来看看finally块何时以及如何执行。代码清单6-7展示了一个示例——它在代码清单6-6的基础上添加了finally代码块。改变的地方使用粗体显示。

代码清单6-7　与try/finally一道工作的yield break

```
static IEnumerable<int> CountWithTimeLimit(DateTime limit)
{
    try
    {
    for (int i = 1; i <= 100; i++)
    {
        if (DateTime.Now >= limit)
        {
            yield break;
        }
        yield return i;
```

① 在未执行yield return或yield break语句之前而离开相关作用域时，它们也会执行。在这里，我只关注这两种yield语句的行为，因为它们的执行流程是不同的。

```
        }
    }
    finally
    {
        Console.WriteLine("Stopping!");      <—— 不管循环是否结束都执行
    }
}
...
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
    Console.WriteLine("Received {0}", i);
    Thread.Sleep(300);
}
```

在代码清单6-7中，如果迭代器块计数到了100或者由于时限而停止，finally代码块都会执行。(如果代码抛出一个异常，它也会执行。)不过，在其他情况下我们可能想避免调用finally块中的代码，我们来走个捷径。

我看到当调用MoveNext时只有迭代器块中的这段代码被执行。那么如果从不调用MoveNext会发生什么呢？或者如果我们只调用几次，而后就停止呢？看看把代码清单6-7的“调用”部分改为下面这样会怎么样：

```
DateTime stop = DateTime.Now.AddSeconds(2);
foreach (int i in CountWithTimeLimit(stop))
{
    Console.WriteLine ("Received {0}", i);
    if (i > 3)
    {
        Console.WriteLine("Returning");
        return;
    }
    Thread.Sleep(300);
}
```

在这里，我们不是提前停止执行迭代器代码，而是提前停止*使用*迭代器。输出结果也许令人感到意外：

```
Received 1
Received 2
Received 3
Received 4
Returning
Stopping!
```

此处，在foreach循环中的return语句执行后，迭代器的finally代码也被执行了。这通常是不应发生的，除非finally代码块被调用了——在这种情况下，被调用了两次！虽然我们知道在迭代器方法中存在着finally代码块，但问题是什么原因引起它执行的呢。

我之前也提到过——foreach会在它自己的finally代码块中调用IEnumerator所提供的Dispose方法（就像using语句）。当迭代器完成迭代之前，你如果调用由迭代器代码块创建的迭代器上的Dispose，那么状态机就会执行在代码当前“暂停”位置范围内的任何finally代码

块。这个解释复杂且有点详细，但结果却很容易描述：只要调用者使用了foreach循环，迭代器块中的finally将按照你期望的方式工作。

只需通过手动使用迭代器，我们就能非常容易地证明对Dispose的调用会触发finally代码块的执行：

```
DateTime stop = DateTime.Now.AddSeconds(2);
IEnumerable<int> iterable = CountWithTimeLimit(stop);
IEnumerator<int> iterator = iterable.GetEnumerator();

iterator.MoveNext();
Console.WriteLine("Received {0}", iterator.Current);

iterator.MoveNext();
Console.WriteLine("Received {0}", iterator.Current);
```

这次，“stopping”行不会打印出来。而如果增加一个对Dispose的显式调用，就会在输出中看到这一行。在迭代器完成之前终止它的执行是相当少见的，并且不用foreach语句而手动使用迭代器也是不多见的，不过如果你这样做，记得把迭代器包含在using语句中使用。

我们现在已经研究了迭代器块的大部分行为了，不过在结束本小节之前，还是值得研究一下在目前微软实现中的一些奇特之处。

6.2.4 具体实现中的奇特之处

如果你使用微软C# 2编译器编译迭代器块，并使用ildasm或Reflector来查看生成的IL，你将看到编译器在幕后为我们生成的嵌套类型。对于我的例子，当编译代码清单6-4的时候，它调用了IterationSample.<GetEnumerator>d__0（顺便说下，在这里的尖括号不是代表泛型类型参数。）我们无法在这里详细地讲解所生成的代码，但应该在Reflector中看看具体发生的情况，建议参考语言规范，在语言规范的10.14节中定义了类型可具有的不同状态，并且其中的描述也有助于理解生成的代码。MoveNext通常包含一个很大的switch语句，将执行大部分工作。

幸好，作为开发人员我们不需要太关心编译器是如何解决这些问题的。不过，关于实现中的以下一些奇特之处还是值得了解的：

- 在第一次调用MoveNext之前，Current属性总是返回迭代器产生类型的默认值；
- 在MoveNext返回false之后，Current属性总是返回最后的生成值；
- Reset总是抛出异常，而不像我们手动实现的重置过程那样，为了遵循语言规范，这是必要的行为；
- 嵌套类总是实现IEnumerator的泛型形式和非泛型形式（提供给泛型和非泛型的IEnumerable所用）。

不实现Reset是完全合理的——编译器无法合理地解决在重置迭代器时需要完成的一些事情，甚至不能判断解决方法是否可行。可以认为，Reset在IEnumerator接口中一开始就不存在，而且我也完全想不起我最后一次调用它是什么时候了。很多集合都不支持Reset，通常，调用者不能依赖它。

实现额外的接口也不会对其产生影响。有意思的是，如果你的方法返回IEnumerable，那么你最终得到的是一个实现了5个接口（包括IDisposable）的类。语言规范解释的很清楚，不过作为开发人员你不需要关心这些。事实上很少有某个类会同时实现IEnumerable和IEnumerator——编译器做了大量的工作，来确保不论你如何处理都正确，通常还会在迭代某个集合时，在创建集合的同一线程上创建一个内嵌类型的实例。

Current的行为有点奇怪——尤其在完成迭代后依然保持着最后的值，会阻止其被垃圾回收。这个问题也许在未来的C#编译器中会被修正，不过这不太可能，因为它会破坏已有的代码（和.NET 3.5及.NET 4一起发布的微软C#编译器还是如此）。严格来说，从C# 2语言规范的角度来看，这样做也是对的——Current属性的行为未明确定义。如果它按照框架文档的建议来实现这个属性，在适当的时候抛出异常可能更好些。

因此，使用自动生成的代码还是存在一些缺点，不过对于明智的调用者，不会有太大问题，让我们正确对待它，毕竟我们节省了大量需要手动实现的代码。换句话说，迭代器应该比在C# 1中使用得更为广泛。下一节提供了一些示例代码，以便检查你对迭代器块的理解，并了解它们在实际的开发中多么有用（而不是仅仅停留在理论上）。

6.3　真实的迭代器示例

你是否曾经写过一些其本身非常简单却能让你的项目更加整齐而有条理的代码？这种事对于我来说是经常发生的，这让我常常得意忘形，以至于同事们都用很奇怪的眼神看我。这种稍显幼稚的快乐，在使用一些新语言特性的时候更加强烈，这不仅仅说明我获得了"玩新玩具"的快感，而是这些新特性显然很好。

即使现在我已经使用了多年的迭代器，仍然会遇到用迭代器块呈现解决方案的情况，并且结果代码简短、整洁和易于理解。本节我将与你分享三个这样的例子。

6.3.1　迭代时刻表中的日期

在开发一个涉及时刻表的项目时，我碰到了几个循环，它们都以如下的代码来开头：

```
for (DateTime day = timetable.StartDate;
     day <= timetable.EndDate;
     day = day.AddDays(1))
```

我处理过太多这样的代码了，一直讨厌这样的循环，不过当我以伪代码的方式向其他开发人员大声念出这些代码的时候，我才意识到我缺乏一定的交流技巧。我会说"对于时刻表中的每一天"。回想起来其实很明显，我实际上是需要一个foreach循环。（这些可能从一开始对你来说就是显而易见的，如果你恰好属于这种情况，那么请原谅我说了这么多。幸好，我看不到你此时的表情。）当重写为下述代码的时候，这个循环就显得好多了：

```
foreach (DateTime day in timetable.DateRange)
```

在C# 1中，我也许只会把它当做一个美梦，而不会自寻烦恼去实现它：我们之前看到要手动

实现一个迭代器有多麻烦，而最后的结果只是使几个for循环变得整洁了一些。然而，在C# 2中，它就非常简单了。在一个表示时刻表的类中，我仅仅添加了一个属性：

```
public IEnumerable<DateTime> DateRange
{
   get
   {
      for (DateTime day = StartDate;
           day <= EndDate;
           day = day.AddDays(1))
      {
         yield return day;
      }
   }
}
```

原始的循环代码被移动到了时刻表类中，这样很好——这些代码被封装到一个对"天"进行循环的属性中，并一次生成1个，这样做比在业务代码中处理这些"天"要好很多。如果我想做得更复杂些（例如，跳过周末和公休假），还可以做到在一个地方封装代码，在任何地方使用它。

6

这个小小的更改，在很大程度上改善了代码库的可读性。因此，这时我停止了对商业代码的重构。我确实也考虑过引入一个Range<T>类型，来表示一个通用的范围，但由于只在这一种情况下需要它，所以为此而付出更多的努力似乎没有什么意义。事实证明，这是一个明智之举。在本书第1版中，我创建了这样一个类型，但它有一些很难在书面上描述的缺点。我在我的工具库中重新设计了这个类型，但仍然还有一些忧虑。这种类型通常听上去要比实际情况简单，然而不久你就需要频繁地处理一些个别情况。我所遇到的这些困难的细节超出了本书的范围，它们大多是对于通用的设计而言的，不是C#本身，但它们却十分有趣，因此我在本书网站上撰写了一篇文章来描述这些情况（参见http://mng.bz/GAmS）。

下面这个例子也是我喜欢的，它能充分说明我衷情迭代器块的原因。

6.3.2　迭代文件中的行

想一想你曾多少次逐行阅读文本文件吧，这实在是一个再平常不过的任务了。而在.NET 4中，框架最终提供了一种方法，使得这项任务通过reader.ReadLines来实现要简单得多。但如果你曾使用过该框架的早期版本，你也可以轻松创建自己的代码，如同我将在接下来的内容中介绍的那样。

我都不知道自己曾经写过多少次这样的代码：

```
using (TextReader reader = File.OpenText(filename))
{
   string line;
   while ((line = reader.ReadLine()) != null)
   {
      // 针对line进行某些操作
   }
}
```

这里共有四个不同的概念：

- 如何获取TextReader；
- 管理TextReader的生命周期；
- 迭代TextReader.ReadLine返回的行；
- 对这些行进行处理。

只有第一条和最后一条是因势而变的——生命周期管理和迭代机制都是样板代码。（至少C#的生命周期管理非常简单，感谢using语句！）我们有两种方法可以进行改进。我们可以使用委托——编写一个工具方法，将阅读器和委托作为参数，为文件中的每一行调用该委托，最后关闭阅读器。这经常作为闭包和委托的示例，不过我还发现了一个更加优雅的适合于LINQ的方法。我们不将逻辑作为委托传入方法，而是使用迭代器一次返回文件中的一行，因此可以使用普通的foreach循环。

你可以编写一个实现了IEnumerable<string>的完整类型来达到这一点（我的MiscUtil库中的LineReader类就是这种用途），不过其他类中的一个单独的方法也是可以的。它非常简单，如代码清单6-8所示。

代码清单6-8　使用迭代器块循环遍历文件中的行

```
static IEnumerable<string> ReadLines(string filename)
{
   using (TextReader reader = File.OpenText(filename))
   {
      string line;
      while ((line = reader.ReadLine()) != null)
      {
         yield return line;
      }
   }
}
...
foreach (string line in ReadLines("test.txt"))
{
   Console.WriteLine(line);
}
```

在对集合进行迭代时，我们将行产生给调用者，除此之外，方法体与之前几乎完全相同。与之前一样，我们打开文件，一次读取一行，然后在结束时关闭阅读器。尽管这里“结束时”这个概念要比在普通方法中使用using语句有趣得多，后者的流控制更为明显。

因此，在foreach循环中释放迭代器非常重要，它可以确保阅读器被清理干净。迭代器方法中的using语句扮演了try/finally块的角色。在到达文件末尾或在中途调用IEnumerater<string>的Dispose方法时，将进入finally块。调用代码很可能滥用ReadLines(...).GetEnumerator()返回的IEnumerator<string>，导致资源泄露，但这通常是IDisposable的情况——如果你没有调用Dispose，则可能导致泄露。不过这很少发生，因为foreach进行了正确的处理。要注意这种潜在的滥用，如果你依赖迭代器中的try/finally块来授予某些权限，然后又将其移除，那么这

就是一个安全漏洞。

这个方法封装了我之前列出的四个概念中的前三个，但却有一些限制。将生命周期管理和迭代部分结合起来是可以的，但如果我们想从网络流中读取文本或使用UTF-8以外的编码格式呢？我们需要将第一部分交还给调用者来控制。最显而易见的方式是修改方法签名，使其接受一个TextReader，如下所示：

```
static IEnumerable<string> ReadLines(TextReader reader)
```

但这是一个糟糕的方案。我们希望获取阅读器的所有权，这样可以方便地为调用者进行清理。但这样一来就意味着，只要用户使用了该方法，我们就不得不进行清理。问题是，如果在第一次调用MoveNext()之前发生了异常，我们就没有机会进行清理了，所有的代码都不会运行。IEnumerable<string>本身不是可释放的，但它已经将这一部分的状态保存为“需要释放”。如果GetEnumerator()被调用两次，还会产生另一个问题：本应生成两个独立的迭代器，但它们却使用相同的阅读器。我们通过将返回类型改为IEnumerator<string>可以在一定程度上缓解这个问题，但这样其结果就无法用于foreach循环了，并且如果我们没有到达第一次调用MoveNext()就出现了错误，则仍然无法运行任何清理代码。幸运的是，有一个办法可以解决这个问题。

因为我们的代码不是立即执行的，因此并不立即就需要阅读器。我们需要的是在需要阅读器的时候获取它的方式。我们可以使用接口来表示“可以在需要的时候提供一个TextReader”，不过含有单一方法的接口总是会让你想到委托。我们这里要小小地做一下弊，使用.NET 3.5中的一个委托。它含有不同参数类型数量的重载，但我们只需要一个：

```
public delegate TResult Func<TResult>()
```

如你所见，该委托没有参数，返回类型与类型参数的类型相同。这是典型的提供器和工厂方法的签名。在本例中，我们想要获取一个TextReader，因此使用Func<TextReader>。对方法的更改非常简单：

```
static IEnumerable<string> ReadLines(Func<TextReader> provider)
{
   using (TextReader reader = provider())
   {
      string line;
      while ((line = reader.ReadLine()) != null)
      {
         yield return line;
      }
   }
}
```

现在，我们只有在需要的时候才去获取资源，并且那时我们处于IDisposable的上下文中，可以在适当的时候释放资源。此外，如果对其返回值多次调用GetEnumerator()，每次都将创建独立的TextReader。

我们可以简单地使用匿名方法来添加打开文件的重载，也可以指定文件的编码：

```
static IEnumerable<string> ReadLines(string filename)
{
   return ReadLines(filename, Encoding.UTF8);
}

static IEnumerable<string> ReadLines(string filename, Encoding encoding)
{
   return ReadLines(delegate {
      return File.OpenText(filename, encoding);
   });
}
```

这个简单的示例使用了泛型、匿名方法（捕获了所在方法的参数）和迭代器块。“三缺一”的是可空类型，否则就汇集了C# 2主要特性的“大四喜”。我曾多次使用过这些代码，它比我们开始时介绍的笨重代码要整洁得多。如同前文提到的那样，如果使用的是.NET的较新版本，你就可以通过`File.ReadLines`来做到。但这仍然可作为一个例子，说明迭代器块可以多么有用。

在最后一个示例中，我们将对LINQ进行尝鲜，尽管我们使用的是C# 2。

6.3.3　使用迭代器块和谓词对项进行延迟过滤

尽管我们尚未开始接触LINQ，但我相信你已经对它有了一定的认识。它允许你用一种简单却强大的方式，对内存集合或数据库等多种数据源进行查询。C# 2没有对查询表达式以及让LINQ变得简洁的Lambda表达式和扩展方法进行任何语言方面的集成，但我们仍然可以实现一些相同的效果。

LINQ的核心特性之一，是使用`Where`方法进行过滤。你提供一个集合和谓词，返回的结果是一个延迟执行的查询，产生集合中只与谓词相匹配的那些项。这有几分像`List<T>.FindAll`，但它是延迟的，并可以与任意`IEnumerable<T>`一起使用。LINQ[①]的一个巧妙之处是将巧妙的部分蕴含于设计之中。实现LINQ to Objects相当简单，我们将证明给你看，至少`Where`方法是这样。颇具讽刺意味的是，尽管大多数让LINQ魅力四射的语言特性都来自C# 3，但它们几乎都是关于如何访问`Where`这样的方法，而不是关于如何实现。

代码清单6-9展示了一个完整的示例，包括简单的参数验证，使用过滤器显示示例代码的源文件中所有`using`指令。

代码清单6-9　使用迭代器块实现LINQ的Where方法

```
public static IEnumerable<T> Where<T>(IEnumerable<T> source,
                                      Predicate<T> predicate)
{
   if (source == null || predicate == null)      <— ❶ 热情地检查参数
   {
      throw new ArgumentNullException();
   }
   return WhereImpl(source, predicate);           <— ❷ 懒惰地处理数据
}

private static IEnumerable<T> WhereImpl<T>(IEnumerable<T> source,
```

① 更准确地说应该是LINQ to Objects。数据库方面的LINQ提供器要复杂得多。

```
                                        Predicate<T> predicate)
{
   foreach (T item in source)
   {
      if (predicate(item))          ❸ 检查当前项与谓词是否匹配
      {
         yield return item;
      }
   }
}
...
IEnumerable<string> lines = LineReader.ReadLines("../../FakeLinq.cs");
Predicate<string> predicate = delegate(string line)
   { return line.StartsWith("using"); };
foreach (string line in Where(lines, predicate))
{
   Console.WriteLine(line);
}
```


如你所见，我们将实现分为两部分：参数验证和真正的过滤逻辑。这有点丑陋，但对于错误处理来说是完全有必要的。假设将所有的内容都放入同一个方法中，那么调用`Where<string>(null, null)`时会发生什么呢？答案是什么也不会发生，或至少不会抛出我们认为的异常。这是由迭代器块的延迟语义所决定的。我们在6.2.2节已经看到，在第一次调用`MoveNext()`之前，不会执行方法体内的任何代码。而你希望的是热情地（eagerly）检查方法的前置条件。异常是不能延迟的，而且这也使调试变得困难。

标准的办法是像代码清单6-9中那样将方法一分为二。首先在普通方法中检查参数❶，然后调用使用迭代器块实现的方法，在得到请求时再延迟处理数据❷。

迭代器块本身十分简单：对于原始集合中的每一项，我们测试谓词❸，如果匹配就生成值。如果不匹配，就测试下一项，直到找到匹配项或迭代完毕。这很简单，但用C# 1实现却很难弄明白（当然还不能有泛型）。

代码的最后部分演示了如何使用这个方法，用之前的示例作为数据，此处就是实现的源代码。谓词简单地测试某一行是否以"using"开头，当然，这可以包含更复杂的逻辑。我为数据和谓词创建了单独的变量，这可以使格式更加清晰，你也可以将它们都写在一行内。我们也可以使用`File.ReadAllLines`和`Array.FindAll<string>`来实现同样的功能，但它与本例最重要的不同是这里的实现是完全延迟和流动的。它每次只需要在内存中保留源文件中的一行即可。当然，本例中的文件很小，可能看不出多大差别，但如果是数十亿字节的日志文件，就能看到这种方法的好处了。

我希望这些示例能让你明白迭代器块为什么如此重要，并且渴望快点学到LINQ的更多内容。在此之前，我要稍微打乱你的思维，为你介绍一个十分奇特（但却相当工整）的迭代器用法。

6.4 使用 CCR 实现伪同步代码

CCR（Concurrency and Coordination Runtime，并发和协调运行时）是微软开发的一个函数库，

为编写适用于复杂的协调情况下的异步代码，提供了另外一种方法。在写作本书的时候，它只是作为微软Robotics Studio（参见http://www.microsoft.com/robotics）的一部分而提供。微软在多个项目中为并发投入了大量的资源，并且.NET 4的并行扩展框架已经成为广大开发者最看重的一个库。但这里我想使用CCR来展示迭代器块如何改变整个执行模型。确实，这种并发使用迭代器块改变执行模型的方法并非巧合，在# 5中为迭代器块而生成的状态机与那些为异步函数而生成的状态机之间的相似性是惊人的。

示例代码处理实际的工作（构建虚拟服务），其理念要比细节重要得多。

假如我们正在编写一个需要处理很多请求的服务器。作为处理这些请求的一个部分，我们需要首先调用一个Web服务来获取身份验证令牌，接着使用这个令牌从两个独立的数据源中获取数据（可以认为一个是数据库，另外一个是Web服务）。然后，我们要处理这些数据，并返回结果。每一个提取数据的阶段都要花费一定的时间——也许要几秒钟。通常我们会考虑简单的同步路由或惯用的异步方法。同步版本可能类似于下面的代码：

```
HoldingsValue ComputeTotalStockValue(string user, string password)
{
   Token token = AuthService.Check(user, password);
   Holdings stocks = DbService.GetStockHoldings(token);
   StockRates rates = StockService.GetRates(token);
   return ProcessStocks(stocks, rates);
}
```

这段代码非常简单和易于理解，不过如果每个请求都要花费2秒钟，那么整个操作将花费6秒钟，并在运行时占用着整个线程。如果我们希望扩展至在并行的模式下处理几十万个请求，那么就会陷入困境。

现在让我们来看一个相当简单的异步版本，在不进行任何处理的时候它不会占用线程[①]，且在可能的地方使用并行调用：

```
void StartComputingTotalStockValue(string user, string password)
{
   AuthService.BeginCheck(user, password, AfterAuthCheck, null);
}

void AfterAuthCheck(IAsyncResult result)
{
   Token token = AuthService.EndCheck(result);
   IAsyncResult holdingsAsync = DbService.BeginGetStockHoldings
      (token, null, null);
   StockService.BeginGetRates(token, AfterGetRates, holdingsAsync);
}

void AfterGetRates(IAsyncResult result)
{
   IAsyncResult holdingsAsync = (IAsyncResult)result.AsyncState;
   StockRates rates = StockService.EndGetRates(result);
   Holdings stocks = DbService.EndGetStockHoldings(holdingsAsync);
   OnRequestComplete(ProcessStocks(stocks, rates));
}
```

① 嗯，大部分情况下，它的效率还是有点低，我们马上就会看到。

这些代码较难阅读和理解，然而，它只是一个简单版本！两个并行调用之间的协调问题只能通过一种简单的方式来解决，因为我们不需要传递其他任何的状态数据，即使如此，这也不是很理想。如果库存服务的调用很快就完成了，我们仍然需要阻塞线程池的线程，等待数据库调用的完成。更重要的是，由于代码在不同的方法之间跳来跳去，导致其中发生的事情不是那么清晰易懂。

到此，你可能会自问，迭代器是如何被利用起来的。好吧，由C# 2提供的迭代器块，实际上可以让你在这个代码块所涉及流程的特定位置“暂停”当前执行过程，并随后携带着同样的状态返回同一位置。设计CCR的聪明人意识到，这正是用于调用*连续传递*（continuation-passing）风格编码所需要的。我们要告知系统，存在哪些需要完成的操作（包括异步地启动其他操作），不过我们可以等待异步操作完成后，再继续执行。我们通过为CRR提供一个`IEnumerator<ITask>`（这里`ITask`是一个由CCR定义的接口）实现来完成这个操作。下面就是我们以这种编程风格实现的请求处理的伪代码：

```
static IEnumerator<ITask> ComputeTotalStockVal.(str.user,str.pass)
{
   string token = null;
   yield return Arbiter.Receive(false, AuthService.CcrCheck(user, pass),
      delegate(string t) { token = t; });

   IEnumerable<Holding> stocks = null;
   IDictionary<string,decimal> rates = null;
   yield return Arbiter.JoinedReceive(false,
      DbService.CcrGetStockHoldings(token),
      StockService.CcrGetRates(token),
      delegate(IEnumerable<Holding> s, IDictionary<string,decimal> r)
        { stocks = s; rates = r; });

   OnRequestComplete(ComputeTotal(stocks, rates));
}
```

被搞糊涂了？我第一次看到这些代码的时候也是如此。不过现在，我对它如此工整的结构惊叹不已。CCR对我们的代码进行了调用（就是在迭代器中对`MoveNext`进行调用），直到第一个`yield return`，我们的语句才会执行。`AuthService`里面的`CcrCheck`方法会启动一个异步请求，CCR会等待（未使用专用线程）直到它完成，最后调用提供给它的委托来处理结果。然后，它会再次调用`MoveNext`，我们的方法就会继续执行。接着，我们并行启动两个请求，在两个请求都完成的时候，CCR就调用另外一个委托来处理它们两个的结果。在这之后，`MoveNext`最后一次被调用，我们就可以完成全部的请求处理。

尽管这比同步版本明显要复杂很多，但仍然把所有东西都写在一个方法当中，按照编写的顺序来执行，并且方法本身也能保持状态（保存在局部变量中，这个变量会在由编译器生成的额外类型中变成状态）。它完全是异步的，它完全摆脱了对线程的直接使用。我没有编写任何错误处理语句，不过它也提供了合理的方式来让你在适当的位置思考出现的问题。

这里我特意没有介绍`Arbiter`类、`ITask`接口等内容的细节。本节的意图不是推广CCR，尽管它读起来和写起来都十分优雅。我认为并行扩展仍然是大多数开发者的首选。我这里的目的是展示迭代器可以用在那些与传统的集合没有丝毫关系的完全不同的上下文环境中。这种用法的核

心思想是状态机：异步开发中两个复杂的问题就是处理状态和在感兴趣的事情发生之前进行有效的暂停。迭代器块使这两个问题得以完美地解决。不过在第15章，你会看到更有针对性的语言支持将使得使代码更整洁。

6.5　小结

C#间接地支持很多模式，你可以用C#实现它们。然而相对而言，只有少数模式是由于作为特别模式下的特定目的的语言特性而被直接支持。在C# 1中，迭代器模式从调用代码的角度来看，是被*直接*支持的，不过从被迭代的集合的角度来看，就并非如此了。编写`IEnumerable`的正确实现是一件枯燥耗时且容易出错的事情。在C# 2中，编译器为你完成了所有无趣的工作，构建了一个状态机来处理迭代器"回调"特性。

应该注意，迭代器块和我们在第5章看到的匿名方法具有共同的方面，即使两者实际的特性大不相同。它们都生成了额外的类型，一些潜在复杂的代码转换被应用到了原始代码上。与C# 1相比，这里的大部分语法糖最明显的例子就是`lock`、`using`和`foreach`。我们将看到，C# 3的几乎每个方面都继续保持着智能编译的这一趋势。

本章还展示了部分LINQ相关的功能：对集合进行过滤。`IEnumerable<T>`是LINQ中最重要的类型之一。如果要编写除LINQ to Objects之外的LINQ操作符①，你会由衷感谢C#团队在语言中添加了迭代器块这一特性。

除了使用迭代器的真实例子外，我们还分析了一个特别的函数库，如何以一种相当奇妙的平时很少使用的方式来使用迭代器，这让我们明白：当我们考虑如何在集合上进行迭代的时候，迭代器还能完成更多的事情。要知道，之前各种各样的语言也都遇到过这样的问题——在计算机科学中，术语"协同程序"（coroutine）用来表示这个特性的概念。而在Unity 3D游戏开发工具箱中，它们也以同样的方式被提及。从历史角度看，各类语言对此或多或少都有支持，有时使用一些技巧也可用来模拟这个特性。——例如，Simon Tatham就有一篇出色的文章讲述了假如你乐意遵循某些编码标准，C语言可以表示协同程序（参见http://mng.bz/H8YX中的"Coroutines in C"一文）。我们也看到了，C# 2让协同程序变得更加容易编写和使用。

在看了一些重要（有时让人痛苦）的几个关键语言特性的改变之后，我们下一章要放缓"改变的步伐"。来描述大量的细小改变，这些改变让C# 2使用起来比起其前辈更加令人愉快。通过对语言的一些琐碎而费时费力的小毛病进行改进，在更大程度上处理了笨拙的向后兼容问题，生成代码终于成为提高工作效率的好帮手。每个特性都相对浅显易懂，不过它们的数量却不少。

① 这远没有听上去那么可怕，而且更有趣。我们将在第12章介绍围绕这个话题的一些准则。

第 7 章

结束C# 2的讲解：最后的一些特性

本章内容

- 分部类型
- 静态类
- 独立取值方法/赋值方法属性访问器
- 命名空间别名
- pragma指令
- 固定大小的缓冲区
- 友元程序集

到目前为止，我们已经学习了C# 2的四大新特性：泛型、可空类型、委托增强和迭代器块。每种特性都是为了满足相当复杂的需求而设计的，因而我们对它们都进行了深入研究。剩下的C# 2新特性只是修整了C# 1的一些小问题。还有一些琐碎的问题是语言设计者决定修正的——某些是由于语言自身的原因需要进行一些改进，还有些是为了使代码生成和本地代码的使用更加方便。

经过一段时间，微软从C#社区（当然也从他们自己的开发人员）那里获得了大量的反馈，了解到了C#的哪些地方还不够“闪亮”。为了减轻这些小小的痛苦，C# 2在那些重大改变之外还做出了一些小的改变。

本章提及的特性都不是特别难懂，所以我们将会快速浏览它们的内容。但是，不要低估它们的重要性——几页纸就能讲完的主题并不意味着它们没有用。你有可能会把它们中的一些特性当作相当常用的基本功能。下面是在本章要提到的这些特性的一个概要，以便你了解后面要讲到的大致内容。

- **分部类型**——可以在多个源文件中为一个类型编写代码。特别适用于部分代码是自动生成，而其他部分的代码为手写的类型。
- **静态类**——对工具类进行整理，以便编译器能明白你是否在不恰当地使用它们，并使你的意图更清晰。
- **独立的取值方法/赋值方法属性访问器**——属性终于有了公有取值方法和私有赋值方法了！

（这不是唯一的组合，不过这是最常见的组合。）

- **命名空间别名**——在类型名称不唯一的情况下的一种解决方式。
- **Pragma指令**——用于操作的特定编译器指令，如禁止对某一特殊代码段使用特定的警告信息。
- **固定大小的缓冲区**——在非安全代码中，可以更多地控制结构处理数组的方式。
- **`InternalsVisibleToAttribute`（友元程序集）**——跨越语言、框架和运行时的一个特性，在需要时能对选定的程序集进行更多的访问。

此时，你也许迫不及待地想了解C# 3的一些让人着迷的特性，我不会怪你的。本章涉及的东西不会让C#一鸣惊人——不过每个特性都可让你的工作更加轻松，或者在某些情况下让你脱离苦海。为了提起大家的兴致，我们第一个要讲述的特性实际上是非常有技巧的。

7.1　分部类型

我们将要看到的第一个改变，是由于在C# 1中使用代码生成器时经常出现的（关于是否应该使用代码生成器的）斗争。对于Windows Forms来说，在Visual Studio中的设计器不得不生成一些开发人员无法操纵的代码，这些用于设计器的代码又和开发人员用来编写用户界面功能的代码放在同一个文件中。这明显是一种很脆弱的方式。

在其他情况下，代码生成器创建了可以和手写代码一同编译的源代码。在C# 1中，可以从自动生成的类上派生一个新类来添加额外的功能，但这是一种很不优雅的方式。因为大量事实表明，在继承链上存在一个不必要的链接，会引发某些问题或降低封装性。例如，如果你的代码中的两个不同部分希望互相调用，那么父类型调用子类型时就需要使用虚方法；反之，子类型调用父类型时，需要使用受保护方法，而其实正常情况下你只需要两个私有的非虚方法。

C# 2允许多个文件为一个类型提供代码，实际上IDE扩展了这一思想，以便用于某个类型的一些代码不用像C#源代码一样完全显示出来。由多个源代码文件组成的类型称为分部类型。

在本节中，我们还将学习*分部方法*，它只能用于分部类型中，可以用丰富而有效的方式把手动书写的钩子代码添加到自动生成的代码中。这实际上是C# 3的一个特性（这次是基于C# 2的反馈作出的改变），不过在我们研究分部类型的时候，与其等到下一部分进行，不如在这里对它进行讨论更符合逻辑。

7.1.1　在多个文件中创建一个类型

创建分部类型是非常容易做的事情——你只需在涉及的每个文件的类型的声明部分附加一个上下文关键字`partial`。虽然本节的所有例子都只用了两个文件，但你可以在任意多个文件中声明一个分部类型。

编译器实际上是在编译之前把所有源文件合并在了一起。这意味着，在一个文件中的代码可以调用另外一个文件中的代码，反之亦然，如图7-1所示，无须“向前引用”或其他技巧。

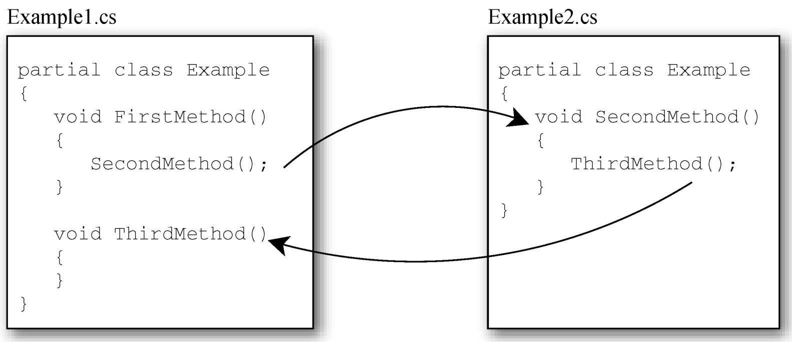


图7-1　在分部类型中的代码能够“看到”类型里的所有成员，不管成员是位于哪个文件中

你不能在一个文件中编写成员的一半代码，而把另外一半代码放到另外一个文件中——每个独立的成员必须完整地位于它所处的文件中。例如，一个方法不能在一个文件中开始，而在另一个文件中结束[①]。对于类型的声明也有一些明显的限制——声明必须要相互兼容。任何文件都能指定要实现的接口（不过不必在那个文件中实现）基类型以及类型参数约束。然而，如果多个文件都设定了基类型，那么它们必须是相同的，并且如果多个文件都设定了类型参数约束，那么约束必须是一致的。代码清单7-1展示了所有的灵活性（虽然它没有完成任何事情，甚至几乎没有什么用处）。

代码清单7-1　演示分部类型的混合声明

```
// Example1.cs
using System;

partial class Example<TFirst, TSecond>
   : IEquatable<string>                    ❶ 接口和类型参数约束
   where TFirst : class
{
   public bool Equals(string other)        ❷ 实现IEquatable<string>
   {
      return false;
   }
}

// Example2.cs
using System;

partial class Example<TFirst, TSecond>
    : EventArgs, IDisposable               ❸ 指定基类和接口
{
   public void Dispose()                   ❹ 实现IDisposable
   {
   }
}
```

① 这有一个例外：分部类型能包含代码跨越同一组文件的嵌套分部类型。

我强调一下，这些代码只是为了讲清楚什么样的声明是合法的——涉及的类型只是由于方便和熟悉才选取的。我们能看到这两个声明（❶和❸）都设定了必须实现的接口。在这个例子当中，每个文件都实现它声明的接口，这也是一种常见的解决方案，不过把`IDisposable`的实现❹移动到`Example1.cs`中，把`IEquatable<string>`的实现❷移动到`Example2.cs`中，也是合法的。我曾使用这个功能将指定的接口与我的实现相分离，将为不同类型生成的相同签名的方法封装到一个接口中。代码生成器并不知道有这样一个接口，因此无法在声明类型时指定其实现了该接口。

只有❶设定了类型约束，且只有❸设定了基类。如果❶要设定基类，那么它必须为`EventArgs`。如果❹要设定类型约束，它必须和❶中设定的完全一致。尤其是，我们不能在❸中为`TSecond`设定类型约束，虽然它并未在❶中有所提及。两个类型必须具有相同的访问修饰符（如果有的话）——例如，我们不能把一个声明为`internal`，另外一个声明为`public`。实际上，围绕组合文件的规则在鼓励一致性的同时，也允许有一定的灵活性。

在“单文件”类型中，成员和静态变量的初始化一定会按照它们在文件中的顺序来发生，不过当使用多文件的时候，就不一定按照这样的顺序发生了。依赖于文件中的声明顺序是脆弱的——如果某个开发人员决定“无恶意地”移动一些代码，那么你的代码就容易出现一些难以捉摸的错误。所以，无论如何你都应该避免这样的情况，尤其要在分部类型中避免它。

既然我们知道哪些能做哪些不能做了，让我们来仔细研究一下为什么要这样做。

7.1.2　分部类型的使用

正如我早先提到的，分部类型主要连接设计器和其他代码生成器。如果代码生成器一定要修改开发人员“拥有”的文件，就会存在出错的风险。利用分部类型模型，代码生成器可以拥有自由操作的文件，或者只要它愿意可以每次都重写整个文件。

某些代码生成器还可以选择不生成任何C#文件，而是等到构建进行的时候再生成。例如，Snippy应用程序就具有一些描述用户界面的XAML（Extensible Application Markup Language，可扩展应用程序标记语言）文件。当项目构建的时候，每个XAML文件都转换为C#文件，并保存到obj目录中（文件名以“.g.cs”结尾以表明它们是被生成的），并同为这个类型提供额外代码（通常是事件处理程序和额外的构造函数代码）的分部类一起编译。这样开发人员就完全不必去修改生成的代码，至少不用去修改超长的构建文件。

除了设计器外，我很小心地使用“代码生成器”这个词语而不是“设计器”，是因为除了设计器外，还存在很多这样的代码生成器。例如，在Visual Studio中，Web服务器代理就生成为一些分部类，并且你也完全可以用自己的工具来基于其他数据源生成代码。一个相当常见的例子就是ORM（Object Relational Mapping，对象关系映射）——一些ORM工具使用来自配置文件（或直接来自数据库）的数据库实体描述信息，并生成表示这些实体的分部类。同样，我的Google Protocol Buffer序列化框架的.NET端口也生成了分部类，已经证明这个特性对于实现本身来说也十分有用。

这使添加行为到类型中变得非常简单：重写基类的虚方法，添加带有业务逻辑的新成员，等等。这是一种非常棒的，让开发人员和工具协同工作的方式，避免了频繁地“争论”谁该负责代码。

如下的方法偶尔会很有用，就是一个生成文件包含多个分部类型，而其中的某些类型在其他文件中得到了增强：为每个类型提供一个手动生成的文件。回到ORM的例子，工具可以生成包含所有实体定义的单个文件，而某些实体能够拥有由开发人员提供的额外代码，为每个实体使用一个文件。这样既保证了自动生成的文件的数量不会太多，也能保证手动编写的代码具有良好的可见性。

图7-2显示了XAML和实体对于分部类型的运用是多么相似，只是在创建自动生成的C#代码的时间选择上稍有不同。

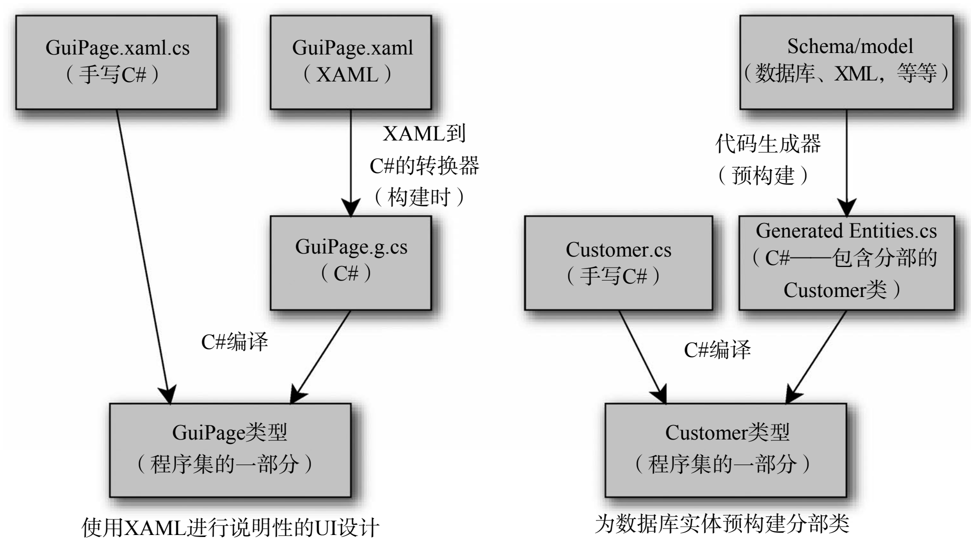


图7-2　XAML预编译和自动生成的实体类的比较

分部类型还有一种稍微不同的用法，就是辅助进行重构。某些时候，一个类型变得太大，担当了太多职责，就应对其进行重构。重构的第一步就是要把这种臃肿的类型分成较小的类，很多关联的内容首先就可以分割为在两个或多个文件上存放的分部类型。接着，我们在文件之间移动方法直到每个文件只处理一种特定的内容，这个工作可以以一种毫无风险的实验方式来完成。在这期间，虽然分割类型的第二步工作不太自动化，不过我们还是很容易看到效果。

分部类还有一个用途，就是单元测试。一个类的单元测试往往会很长，甚至超出了实现本身。可以使用分部类型将测试划分为多个易于理解的小块。你仍然可以一次运行某类型的所有测试（因为我们仍然只有一个测试类），然而我们却可以在不同的文件中查看针对不同功能的测试，就像Visual Studio生成分部类型时那样。通过手工修改项目文件，你还可以在Solution Explorer中看到相同的父/子扩展。这可能并不适合所有人，但很适合我，我发现用这种方法管理测试十分方便。

当分部类型首次在C# 2中出现的时候，没人确切知道要如何使用它们。人们迫切需要的特性是一种方式，能为生成的方法提供可以调用的“额外”代码。C# 3提供的分部方法满足了这种需求。

7.1.3　C# 3 独有的分部方法

再重申一下之前的说明，虽然本章该部分的其他内容讲解的是C# 2特性，但分部方法不适合和其他任何C# 3特性一起讲解，它们更适合在描述分部类型的时候进行讲述。很抱歉，这可能会让大家感到混乱。

回到这个特性，有些时候，我们希望能在手动创建的文件中指定某种行为，并在自动生成的文件中使用该行为。例如，在一个具有很多自动生成属性的类中，我们希望能够指定执行代码，作为某些属性接受新值的验证。另外一个常见的情况是，为代码生成工具包含构造函数——手写的代码经常想挂钩到对象构造过程中，以设定默认值，执行某些日志记录，等等。

在C# 2中，这样的需求只能通过使用能够订阅事件的手动生成代码，或者通过创建自动生成代码来满足（假设手写代码中包括某些具有特别名称的方法）——如果这些相关方法未提供，那么整个代码都无法通过编译。还有其他可选的方法，如生成代码可以提供一个基类，该基类具有一些默认情况下不完成任何事情的虚方法。手动生成的代码就可以从这个类派生，并覆盖部分或全部方法。

所有这些解决方案都有点麻烦。C# 3的分部方法提供了一些可选的钩子，如果没有实现它们，也不会产生任何开销——任何对未实现的分部方法的调用，都会被编译器移除。因此你可以广泛使用工具提供的钩子，只有出现在编译代码里的东西才会有所开销。

通过一个例子，很容易理解这个功能。代码清单7-2显示了一个保存于两个文件中的分部类型，自动生成的代码中的构造函数调用了两个分部方法，手动生成的代码实现了其中的一个。

代码清单7-2　分部方法在构造函数中被调用

```
// Generated.cs
using System;
partial class PartialMethodDemo
{
   public PartialMethodDemo()
   {
      OnConstructorStart();
      Console.WriteLine("Generated constructor");
      OnConstructorEnd();
   }

   partial void OnConstructorStart();
   partial void OnConstructorEnd();
}

// Handwritten.cs

using System;
partial class PartialMethodDemo
{
   partial void OnConstructorEnd()
   {
      Console.WriteLine("Manual code");
   }
}
```

正如代码清单7-2所示，分部方法的声明方式与抽象方法相同：只使用partial修饰符提供签名而无须任何实现。同样，实际的实现还需要partial修饰符，不然就和普通方法一样了。

对PartialMethodDemo的无参构造函数进行调用，输出结果为“Generated constructor”，接着“Manual code”也会被打印出来。分析构造函数的IL，你不会看到对OnConstructorStart的调用，因为它已经不存在了——在这个编译好的类型中，没有它的任何痕迹。

由于方法可能不存在，分部方法返回类型必须为void，且不能获取out参数。它们必须是私有的，但可以是静态的或泛型的。如果方法没有在任何文件中实现，那么整个调用语句就会被移除，包括任何参数计算。如果任何你打算进行的参数计算具有副作用，那么你应该单独执行这些计算，不管分部方法是否实现。例如，假设你有如下代码：

```
LogEntity(LoadAndCache(id));
```

这里，LogEntity是一个分部方法，而LoadAndCache从数据库加载一个实体，并将其插入缓存中。你应该使用下面的代码：

```
MyEntity entity = LoadAndCache(id);
LogEntity(entity);
```

使用这种方法，无论是否为LogEntity提供了实现，实体都可以加载并缓存。当然，如果实体能够在后面同样“廉价”地载入，或者可能根本不需要，那么就应该使用第一种形式，以避免在某些情况下出现不必要的负荷。

老实说，除非你编写自己的代码生成器，否则你可能更多地是去实现分部方法，而不是去声明并调用它们。如果你只是实现它们，就无须担心参数计算的问题。

概括来说，C# 3的分部方法让生成代码可以和手写代码以一种丰富的方式进行交互，而在无须这种交互的情况下不会产生任何性能损失。C# 2分部类型可以让代码生成工具和开发人员之间的关系更富有成效，而分部方法是对分部类型的一种自然延续。

我们下面要研究的特性大不一样，它通过某种方式来告知编译器一个类型预期特性的更多信息，以便可以在类型本身和使用这个类型的代码上执行更多的检查。

7.2 静态类型

我们的第2个新特性在某种意义上说完全没有必要——它只是在编写工具类的时候，让代码整齐一些，稍微优雅一些的一种方式。

每个人能都会编写工具类。不管在Java还是C#中，我从未看到，在一个重大项目中不包含任何只由静态方法构成的类。罪经典的例子就是包含有字符串辅助方法的类型，完成一些转义、反向、智能替换——凡是你想得起的操作。Framework的一个例子是System.Math类。工具类的主要特性如下：

- 所有的成员都是静态的（除了私有构造函数）；
- 类直接从object派生；
- 一般情况下不应该有状态，除非涉及高速缓存或单例；

- ❑ 不能存在任何可见的构造函数；
- ❑ 类可以是密封的（添加sealed修饰符），当然要开发人员记得这样做。

最后两项是可选的，而且实际上如果不存在可见的构造函数（包括受保护的），那么类实际上也就是密封的了。不过，这两项都是为了让类的目的更加明确。

代码清单7-3给出了一个C# 1工具类的例子——接着我们会看到C# 2如何改进其中的问题。

代码清单7-3　典型的C# 1工具类

```
public sealed class NonStaticStringHelper          ❶ 密封类以防止派生
{
    private NonStaticStringHelper()                ❷ 防止其他代码对其进行实例化
    {
    }

    public static string Reverse(string input)     ❸ 所有的方法都是静态的
    {
        char[] chars = input.ToCharArray();
        Array.Reverse(chars);
        return new string(chars);
    }
}
```

该类是封闭的❶，因此不能派生子类。继承应该是一种特殊化，然而此处没有任何可特殊化的东西，因为除了私有的构造函数❷之外，所有的成员都是静态的❸。这个构造函数第一眼看上去有点奇怪，既然它是私有并且从来不被使用，那么为何要添加它呢？原因是，如果你不为类提供任何构造函数，那么C# 1编译器总是会提供一个公有的*默认的无参构造函数*。在这种情况下，我们不希望出现任何可见的构造函数，所以不得不提供一个私有的。

这种模式起到了相当好的作用，不过C# 2让其变为显式并积极防止这个类型被误用。我们首先看到要进行改变的地方是，当在C# 2中定义时，需要把代码清单7-3中的内容调整为“适当的”静态类。正如你从代码清单7-4中所看到的，只需一点点修改就行。

代码清单7-4　将代码清单7-3中的工具类转化为一个C# 2静态类

```
using System;

public static class StringHelper
{
   public static string Reverse(string input)
   {
      char[] chars = input.ToCharArray();
      Array.Reverse(chars);
      return new string(chars);
   }
}
```

这次，我们在类声明中使用了static修饰符而不是sealed，而且我们不用包含任何构造函数——这些就是仅有的代码差异。C# 2编译器知道静态类不用包含任何构造函数，所以它也不会提供默认的。

实际上，编译器在类定义上执行了大量的约束：

- 类不能声明为abstract或sealed，虽然两者都是隐含声明的；
- 类不能设定任何要实现的接口；
- 类不能设定基类型；
- 类不能包含任何非静态成员，包括构造函数；
- 类不能包含任何操作符；
- 类不能包含任何protected或protected internal成员。

应当注意，即使所有成员都必须为静态的，你还是要把它们都显式地声明为静态的，除了嵌套类型和常量。虽然嵌套类型是外围类的隐式静态成员，不过如果不要求的话，嵌套类型本身可以不用是静态的。

编译器不仅在静态类的定义上设置了很多约束，而且还保证它不会被误用。我们知道，不能实例化这种类，任何尝试实例化它的操作都会被禁止。例如，在StringHelper是静态类的时候，下面所有这些语句都是无效的：

```
StringHelper variable = null;
StringHelper[] array = null;
public void Method1(StringHelper x) {}
public StringHelper Method1() { return null; }
List<StringHelper> x = new List<StringHelper>();
```


如果这个类遵循C# 1的模式，那么上述的语句都可以使用，不过它们都基本上没有什么用。简而言之，C# 2中的静态类不允许你做任何以前就不允许做的事情——当然，无论如何只是禁止了一些你不应该做的事情。它们还显式地表明了你的意图。让类成为静态的，就是在说你绝对不会创建该类的任何实例。这不是一个实现方面的技巧，而是一种设计选择。

我们列表中的下一个特性更具有积极的意义。它用于一个比较特殊却有十分常见的场景，不过与C# 1的方案不同的是，它既不难看也不会破坏封装。

7.3 独立的取值方法/赋值方法属性访问器

我承认，当第一次看到C# 1不允许为属性设定一个公开的取值方法和一个私有的赋值方法的时候，我感到有点郁闷。它不是唯一被C# 1禁止了的访问器修饰符组合，但它是最常见的一种需求方式。实际上，在C# 1中取值方法和赋值方法必须具有相同的可访问性——它作为属性声明的一部分进行声明，而不是作为取值方法或赋值方法声明的一部分进行声明的。

希望取值方法和赋值方法具有不同的可访问性的充分理由是——通常，在改变属性支持的变量时，你可能需要进行一些验证、日志记录、锁定或者执行其他代码，但是你不希望让属性对于类的外部代码是可写的。在C# 1中，要么破坏封装让属性变为公共可写，要么在类中编写一个SetXXX()方法来进行赋值。前者违背我们的意图，后者在你已经习惯使用“真正”的属性后，会让你感觉有点丑陋。

C# 2通过为取值方法或赋值方法显式地设置更多的访问限制，使其不受属性本身声明的约束。通过一个例子就很容易了解这种方式：

```
string name;

public string Name
{
   get { return name; }
   private set
   {
      // Validation, logging etc here
      name = value;
   }
}
```

在这个例子当中，Name属性实际上对于其他所有类型[①]都是只读的，不过我们能在类型内部，使用熟悉的属性语法来设置该属性。这种语法也适合于索引器和其他属性。你可以让赋值方法比取值方法的可见性更高（例如，一个受保护的取值方法和一个公有赋值方法），不过这种情况很少见。同样，只写属性比其只读属性也要少见得多。

说明　**一些琐碎的事情：只有这个地方是要求明确指出“私有”的**　在C#的其他地方，在给定的情况下，默认的访问修饰符可能大部分都是私有的。换句话说，如果某些内容能被声明为私有，那么省略的访问修饰符就被完全默认为私有。这是一种很好的语言设计元素，因为这样很难意外地发生错误：如果你希望某些内容更加公开，在你使用它的时候会注意到。但如果你意外地让某些内容“太公开”，那么编译器无法帮助你辨认出问题所在。但属性取值方法或赋值方法访问器的设定就是个例外——如果你不设定任何东西，那么默认情况下取值方法/赋值方法和属性本身整体上保持一致的访问修饰符。

还要注意，你不能把属性本身声明为私有，而让取值方法是公有的——你只能设置比属性更私有的特殊取值方法/赋值方法。而且，你不能为取值方法和赋值方法同时设定一个访问修饰符——说起来有点好笑，因为你可以将属性本身声明为两个修饰符中更公有的一个。

针对封装的增强功能一直未兑现。可惜，在C# 2依然没有阻止在同一个类中的其他代码绕过属性直接访问它支持的字段。我们将在下一章中看到，C# 3在特殊的情况下修正了这个问题，不过在通常的情况还是未能解决。

现在，我们要从一个你很可能想经常使用的特性，转到一个大部分时间你都想避免的特性——它允许你的代码在涉及它所属的类型时能被完全地显现出来，不过这样做提高了阅读代码的难度。

7.4　命名空间别名

命名空间的主要意图是将众多类型组织成一个有用的层级。它们还允许你在类型的非限定名可能相同的时候用完全限定名来保持类型的唯一性。当然，这并不是说没有充分的理由就要重复使用非限定类型名，但有时，这却是自然而然的事情。

① 除了嵌套类型，它总是能访问外围类型的私有成员。

例如非限定名Button。在.NET 2.0 Framework中，有两个类的名字是：System.Windows.Forms.Button和System.Web.UI.WebControls.Button。虽然它们都叫Button，但通过命名空间很容易区分。这和真实生活很类似——你也许知道好几个叫Jon的人，但你不太可能知道其他叫Jon Skeet的人。如果你和朋友在特定环境下交谈，你很可能只用Jon这个名字而不用确切地说出这个人的全名——但在其他情况下，你可能需要提供更确切的信息。

C# 1的using指令（不要和自动调用Dispose的using语句混淆）能够用于两种情况——一种是为命名空间和类型创建一个*别名*（例如，using Out=System.Console;），另外一种就是将一个命名空间引入到当编译器查找某个类型（例如，using System.IO;）时可以搜索到的上下文列表中。总的来说，这已经足够了，不过还是有少数几个语言无法处理的情况，以至于在自动生成代码的地方，不得不弃用这种方式，以绝对保证无论发生什么，都使用正确的命名空间和类型。

C# 2解决了这些问题，增加了语言的健壮性。不仅生成的代码在执行的时候更加健壮，而且无论是类型、程序集或被导入的命名空间，你都可编写代码来确保它按照你的意图执行。这些非常手段很少在自动生成代码的外部用得上，不过还是有必要了解一下，以便在需要的时候可以用得上。

在C# 2中，有3种别名种类：C# 1的命名空间别名、全局命名空间别名和外部别名。我们从已经存在于C# 1中的别名类型入手，引入一种使用别名的新方法，来确保编译器知道把它看做是别名而不是把它当作另一个命名空间或类型的名称去检查。

7

7.4.1　限定的命名空间别名

就算在C# 1中，也应该尽可能避免使用命名空间别名。偶尔，你或许会发现两个类型名称有冲突（正如之前的Button的例子那样），那么，你要么每次使用它们的时候都设定包含了命名空间的完整名称，要么取别名并以一种类似命名空间缩写的方式来区分二者。代码清单7-5演示了两个Button类型通过别名来限定和使用的例子。

代码清单7-5　使用别名来区分不同的Button类型

```
using System;
using WinForms = System.Windows.Forms;
using WebForms = System.Web.UI.WebControls;

class Test
{
   static void Main()
   {
      Console.WriteLine(typeof(WinForms.Button));
      Console.WriteLine(typeof(WebForms.Button));
   }
}
```

代码清单7-5在编译时不会出现任何错误或警告，虽然如此，不过如果一开始我们只需处理一种类型的Button，那么这种方法还是不能尽如人意。然而还有这样一个问题：如果某人想引入名为WinForms或WebForms的类型或命名空间，又该如何处理呢？编译器无法知道WinForms.Button是什么意思，且有可能优先于别名而使用类型或命名空间。我们希望能够告

知编译器，即使WinForms已经出现在其他地方，还是需要它把它当作一个别名来处理。

C# 2引入了“::”命名空间别名修饰符语法来完成这个工作，如代码清单7-6所示。

代码清单7-6　使用::来告知编译器使用别名

```
using System;
using WinForms = System.Windows.Forms;
using WebForms = System.Web.UI.WebControls;

class WinForms {}

class Test
{
   static void Main()
   {
      Console.WriteLine(typeof(WinForms::Button));
      Console.WriteLine(typeof(WebForms::Button));
   }
}
```

代码清单7-6使用WinForms::Button，而不是WinForms.Button，这样编译器就明白要干什么了。如果你把“::”改回“.”，将会出现一个编译错误。

那么，如果你在任何要使用别名的地方使用“::”，就没什么问题了，对吗？哦，也不完全是这样……

7.4.2　全局命名空间别名

你无法为命名空间层级的根或全局命名空间定义别名。假设你有两个类，都叫Configuration。一个位于MyCompany命名空间内部，另外一个根本未指定命名空间。现在，你如何在MyCompany命名空间内引用“根”Configuration类？你不能使用普通的别名，并且如果你仅仅指定Configuration，那么编译器将使用MyCompany.Configuration。

在C# 1中，没有完全解决这个问题的方法。C# 2再次“解救了大家”，允许你使用global::Configuration来告知编译器你确切需要的东西。代码清单7-7演示了遇到的问题和解决方法。

代码清单7-7　使用全局命名空间别名来准确地设定想要的类型

```
using System;

class Configuration {}

namespace Chapter7
{
   class Configuration {}

   class Test
   {
      static void Main()
      {
         Console.WriteLine(typeof(Configuration));
         Console.WriteLine(typeof(global::Configuration));
         Console.WriteLine(typeof(global::Chapter7.Test));
      }
   }
}
```

代码清单7-7中大部分代码只是设立了这样一种状况——Main内的这3行代码才是我们所关注的。第1行会打印“Chapter7.Configuration”，因为编译器在移动到命名空间根之前把Configuration解析为这个类型。第2行表明，类型必须位于全局命名空间中，所以只打印出“Configuration”。我包含第3行只是为了演示，利用全局命名空间，你依旧能够引用命名空间内部的类型，但要设置完全限定名。

此时，只要有必要就使用全局命名空间，我们总是能得到任意的唯一命名类型——实际上，如果你在对代码可读性没有要求的位置编写一个代码生成器,那么你可能希望随意地使用这个特性来保证你总是指向正确的类型，而无论其他类型在代码编译的时候实际上表示的是什么。那么，即使当我们包含了类型的命名空间，它的名称依然不唯一，我们该怎么办？这确实越来越复杂了……

7.4.3 外部别名

在本节开始的地方，我用人名来比喻命名空间和上下文。我特别指出，你不太可能知道很多个叫Jon Skeet的人。然而，我知道不止一个人和我同名，假如你知道两个或多个和我同名的人，也不足为奇。在这种情况下，为了明确你意指的是哪个，你必须提供比全名更多的信息——你认识这个人的原因，他所在的国家，或者类似的某些与众不同的特点。

7

在C# 2中，你可以使用外部别名来指定这些额外信息——这个名称，不仅存在于你的源代码中，还存在于你传递给编译器的参数中。对于微软C#编译器，你需要指定类型所在的程序集。假设有两个程序集First.dll和Second.dll，都包含了名为Demo.Example的类型。我们无法仅使用完全限定名区分它们，因为它们的完全限定名相同。但能使用外部别名来指定我们意指的是哪个。代码清单7-8演示了一个相关的代码，并附带了用于编译它的命令行指令。

代码清单7-8 调用在不同程序集中，具有相同名称的不同类型

```
// Compile with
// csc Test.cs /r:FirstAlias=First.dll /r:SecondAlias=Second.dll

extern alias FirstAlias;                         ❶ 指定两个外部别名
extern alias SecondAlias;

using System;
using FD = FirstAlias::Demo;                     ❷ 使用命名空间别名来使用外部别名

class Test
{
   static void Main()
   {                                             ❸ 使用命名空间别名
      Console.WriteLine(typeof(FD.Example));
      Console.WriteLine(typeof(SecondAlias::Demo.Example));   ❹ 直接使用外部别名
   }
}
```

在代码清单7-8中的代码非常直观。首先，我们需要引入两个外部别名❶，表示你既能直接使用外部指令❹，也可以通过命名空间别名（❷和❸）来使用。实际上，没有别名的普通using指令（例如，using FirstAlias::Demo;）允许我们使用名称Example，而根本无须任何进一

步的限定。也要注意，一个外部别名可以涵盖多个程序集，且多个外部别名也可以都指向同一个程序集，不过，使用这些特性必须十分谨慎，尤其是组合使用的情况。

如果要在Visual Studio中设定外部别名，只需在Solution Explorer里选择一个程序集引用，并在Properties窗口中编辑Aliases值，如图7-3所示。

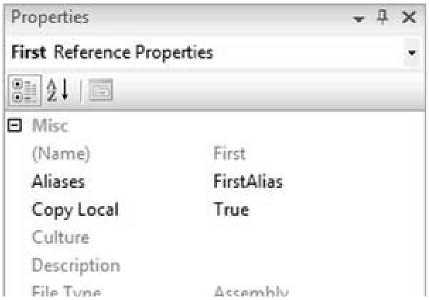


图7-3　Visual Studio 2010的Properties窗口的一部分，显示了First.dll引用的外部别名FirstAlias

当你需要使用这种方案的时候，尽管用。必要时也可以使用偶然使用了相同的完全限定类型名称的第三方程序集。那些你对命名控制很严格的地方，反而可以保证你的名称不会在这样的地方出现问题。

我们下一个特性相当于一个元特性（meta-feature）。它达到的确切目的依赖于你使用的编译器，因为它的目的就是为了能够控制编译器特定的特性——不过我们只关注微软编译器。

7.5　pragma 指令

对pragma指令进行描述通常都非常简单：pragma指令就是一个由#pragma开头的代码行所表示的预处理指令，它后面能包含任何文本。pragma指令的结果无法把程序行为改为违反C#语言规范的任何东西，不过它可以实现规范之外的所有事情。如果编译器不理解某个特别的pragma指令，那么只会发出一个警告而非错误。

规范的每样东西基本都是围绕这个主题展开的。微软C#编译器理解两种pragma指令：警告（warning）和校验和（checksum）。

7.5.1　警告pragma

只有少数情况下，C#编译器才会发出有理但烦人的警告。对编译器警告的正确反应，应该是去修复它——通过修复警告，代码通常会得到改进，而不是变得更糟。

但有时你需要忽略一个警告，这时警告pragma就派上用场了。例如，我们创建了一个从未读取和写入的私有字段。它几乎总是没用的……除非我们碰巧知道它将通过反射来使用。代码清单7-9用一个完整的类演示了这种情况。

代码清单7-9 包含了未用字段的类

```
public class FieldUsedOnlyByReflection
{
    int x;
}
```

如果你尝试编译代码清单7-9，将获得一个类似这样的警告消息：

```
FieldUsedOnlyByReflection.cs(3,9): warning CS0169:
The private field 'FieldUsedOnlyByReflection.x' is never used
```

这是来自命令行编译器的输出。在Visual Studio的Error List窗口中，你也能看到同样的信息（加上它所在的项目），但你无法得到警告编号（CS0169）。为了找出这个编号，你需要选中这个警告并显示和它关联的帮助信息，或者在显示了完整文本的Output窗口中查看，在这里。我们需要这个编号来实现没有警告的代码编译，如代码清单7-10所示。

代码清单7-10 禁用（和恢复）警告 CS0169

```
public class FieldUsedOnlyByReflection
{
#pragma warning disable 0169
    int x;
#pragma warning restore 0169
}
```

代码清单7-10是不需加以说明的——第1个pragma禁用了我们感兴趣的特别警告，第2个恢复了它。禁用尽可能小的一段代码的警告是一个良好的做法，以便你不会错过任何真正应该修复的错误。如果你想在单独一行上禁用或恢复多个警告，那么只需用逗号分隔多个警告编号。如果你没有指定任何警告编号，将一次性禁用或恢复所有警告——不过无论从哪个方面想这都是个坏主意。

7.5.2 校验和pragma

你不太可能需要微软编译器能识别的第2种pragma形式。它支持调试器在源代码文件中找到特定的位置。通常，当C#文件被编译的时候，编译器生成一个来自于文件的校验和，并把它包含到调试信息中。当调试器需要定位源代码文件并寻找多个可能的匹配的时候，它能为每个候选文件生成校验和，并检查哪个是正确的。

现在，当ASP.NET页面转换为C#的时候，C#编译器看到的是生成文件。生成器计算.aspx页面的校验和，并使用校验和pragma来告知C#编译器，使用这个校验和而不是基于生成页面再计算一个。

校验和pragma的语法如下：

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

GUID指出用于计算校验和的散列算法是哪个。`CodeChecksumPragma`类的文档给出了SHA-1和MD5的GUID，在你希望实现自己的具有调试器支持的动态编译框架的时候，可能用得上。

C#编译器未来的版本可能将包括更多的pragma指令，并且其他的编译器（比如Mono编译器，gmcs）也可以具有它们自己的支持不同特性的指令。翻阅编译器文档可以获取最新的信息。

下一个特性是另外一个你大概不太可能用得到的东西——但同时，如果你使用它，或许能让你的工作更加轻松。

7.6　非安全代码中固定大小的缓冲区

在使用P/Invoke调用本地代码的时候，自行处理被定义具有特定长度的缓冲区的结构，是很平常的。在C# 2之前，类似的结构很难直接处理，即使通过非安全代码也是如此。现在，你可以声明一个大小合适的缓冲区，直接嵌入到这个结构的数据当中。

这个功能不仅仅用于调用本地代码，尽管这是它最主要的用途。例如，使用它可以很容易地填充直接和文件格式对应的数据结构。语法很简单，我们再次用一个例子来演示它的用法。为了创建一个字段，来把一个20字节的数组嵌入到它的外围结构中，可以使用如下语句：

```
fixed byte data[20];
```

这段代码把data当作一个byte*（指向字节数据的指针）来使用，虽然整个内部实现中，实际上是C#编译器在声明的类型内部创建一个新的嵌套类型，并在变量本身上应用新的FixedBuffer特性标记。接着CLR适当地处理内存分配问题。

这个特性的一个不足之处是，它只能在非安全代码中才可使用：外围的结构必须在一个非安全环境中声明，并且你只能在非安全环境中使用固定大小的缓冲区成员。这限制了它可以发挥作用的地方，不过必要时它还是一种可用的好方法。同时，固定大小的缓冲区只能应用于基元（primitive）类型，但无法作为类成员（只能是结构成员）来使用。

有几个值得注意的Windows API，这个特性可以在其中直接发挥作用。最常遇到的情况是，需要一个固定长度的字符数组（例如，TIME_ZONE_INFORMATION结构），然而，由于封送处理器（marshaler）的阻碍，字符的固定大小的缓冲区似乎无法被P/Invoke处理得很好。

我们来看下面这个例子，代码清单7-11是一个控制台应用程序，在当前控制台窗口中显示了可用的颜色。它使用了API函数GetConsoleScreenBufferEx，它是Vista和Windows Server 2008中的一个新函数，获取扩展的控制台信息。代码清单7-11以十六进制的格式（bbggrr）显示了全部的16种颜色。

代码清单7-11　演示获取控制台颜色信息的固定大小的缓冲区

```
using System;
using System.Runtime.InteropServices;

struct COORD
{
   public short X, Y;
}

struct SMALL_RECT
{
```

```
    public short Left, Top, Right, Bottom;
}

unsafe struct CONSOLE_SCREEN_BUFFER_INFOEX
{
    public int StructureSize;
    public COORD ConsoleSize, CursorPosition;
    public short Attributes;
    public SMALL_RECT DisplayWindow;
    public COORD MaximumWindowSize;
    public short PopupAttributes;
    public int FullScreenSupported;
    public fixed int ColorTable[16];
}

static class FixedSizeBufferDemo
{
    const int StdOutputHandle = -11;

    [DllImport("kernel32.dll")]
    static extern IntPtr GetStdHandle(int nStdHandle);

    [DllImport("kernel32.dll")]
    static extern bool GetConsoleScreenBufferInfoEx
        (IntPtr handle, ref CONSOLE_SCREEN_BUFFER_INFOEX info);

    unsafe static void Main()
    {
        IntPtr handle = GetStdHandle(StdOutputHandle);
        CONSOLE_SCREEN_BUFFER_INFOEX info;
        info = new CONSOLE_SCREEN_BUFFER_INFOEX();
        info.StructureSize = sizeof(CONSOLE_SCREEN_BUFFER_INFOEX);
        GetConsoleScreenBufferInfoEx(handle, ref info);

        for (int i=0; i < 16; i++)
        {
            Console.WriteLine ("{0:x6}", info.ColorTable[i]);
        }
    }
}
```

代码清单7-11使用固定大小的缓冲区来保存颜色表。在固定大小的缓冲区出现之前，我们仍可以为每个颜色表入口的一个字段使用API，或者通过把一个普通的数组封送为UnmanagedType.ByValArray来使用API。然而，这样做就在“内存堆”上创建了一个单独的数组，而不是在结构中保存所有信息。在这里，不会出现问题，不过在一些高性能的情形下，最好还是把数据“堆放”在一起。与性能有关的另一件事是，如果缓冲区是托管内存堆上的数据结构的一部分，那么你必须在访问它之前“钉住”它。如果你频繁这样做，就会极大地影响垃圾收集器。当然，基于栈的结构没有这个问题。

我不认为固定大小的缓冲区是C# 2中的一个非常重要的特性——至少，它对于大多数人都不重要。然而为了完整性，我把它包含在本书中，但是毫无疑问某些人在某个地方会发觉它是多么的有用。

我们最后一个特性几乎不能算是C# 2特性，不过也能勉强作数，所以还是为了完整性，我把它也包含在本书中。

7.7　把内部成员暴露给选定的程序集

有一些特性，在语言中比较明显——例如，迭代器块。有一些特性对于运行时比较明显，比如JIT编译器优化。还有一些特性显然是“脚踏两只船”，如泛型。这个最后的特性和每个方面都搭点边，不过又是十分古怪，在任何规范中都不屑去提及它。另外，它使用了一个在C++和VB.NET中具有不同的意思的术语——现在它添加了第3个意思。公平地说，在访问权限上下文中使用的所有术语，都具有不同的含义。

7.7.1　简单情况下的友元程序集

在.NET 1.1中，准确地说定义为内部的类型、方法、属性、变量或事件，都只能在其声明的同一个程序集中被访问到，这么说是完全正确的[①]。在.NET 2.0中，依旧大致还是如此，不过提供了一个新的属性来让你稍稍打破规则：`InternalsVisibleToAttribute`，通常就是指`Internals-VisibleTo`。（当应用一个名称结尾为`Attribute`的属性时，C#编译器将自动地添加后缀。）

`InternalsVisibleTo`只能用于程序集（而非特定的成员），并且你可以在同一个程序集中应用多次。我们将把包含这个属性的程序集称为*源程序集*（source assembly），当然这完全是非官方的术语。在你应用这个属性的时候，你必须设定另外一个程序集，即通常所说的*友元程序集*（friend assembly）。结果，友元程序集能够看到源程序集的所有内部成员，就如同它们是公有的一样。这可能听起来令人感到担忧，不过它还是有用的，我们接着就会看到。

代码清单7-12用属于3个不同程序集的3个类来展现了这个功能。

代码清单7-12　友元程序集的演示

```
// Compiled to Source.dll
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("FriendAssembly")]        <-- 授予额外的访问权限
public class Source
{
   internal static void InternalMethod() {}

   public static void PublicMethod() {}
}

// Compiled to FriendAssembly.dll
public class Friend
{
   static void Main()
   {
      Source.InternalMethod();          <-- 在FriendAssembly中使用额外的访问权限
      Source.PublicMethod();
   }
}
```

① 当以适当的权限运行时，不包括在适当的许可下使用反射的情况。

```
// Compiled to EnemyAssembly.dll
public class Enemy
{
   static void Main()
   {
      // Source.InternalMethod();        ❶ EnemyAssembly不具备特殊的访问权限
      Source.PublicMethod();           依旧可以正常访问公有方法
   }
}
```

在代码清单7-12中，FriendAssembly.dll和Source.dll之间存在一种特殊的关系，然而只能单向操作：Source.dll不能访问FriendAssembly.dll中的内部成员。如果我们取消在❶行的注释，Enemy类将编译出错。

那么，究竟是为什么，我们想要把精心设计的程序集向某些程序集开放呢？

7.7.2 为什么使用InternalsVisibleTo

我没有在两个产品程序集之间使用过InternalsVisibleTo。我不是说不存在合法的使用情况来使用它，只是我尚未遇到过这种情况。然而，我在进行单元测试的时候使用过这个属性。

有一些人说，你只应该测试代码的公开接口。就我个人看来，我很高兴做哪些以尽可能简单方式进行的测试。友元程序集让事情变得非常容易：使测试那些只能进行内部访问的代码的工作，刹那间变成了"小菜一碟"，而不用再经历由于测试的目的而使成员变为公有这样的不确定的步骤了，也不用再把测试代码包含到产品程序集内部了。（这有时意味着，为了测试目的就让成员为内部的，否则就让它们为私有的，不过不用太担心这个过程。）

唯一的缺点是，你的测试程序集名称会保留在你的产品程序集中。理论上，如果你的程序集未签名，并且你的代码通常在受限权限集下运作，那么这可能表示着一个安全攻击向量。（首先，具有全信任权限的任何人都可以使用反射来访问这些成员。你自己能用这种方法来完成单元测试，不过这样做不太好。）如果任何人认为这是一个真正严重的问题，而弃用这个特性的话，我会感到非常吃惊的。它确定不太好，所以现在引入签名程序集这个法宝。只有当你认为它是一个很好且比较简单的特性……

7.7.3 InternalsVisibleTo和签名程序集

如果一个友元程序集是签名的，那么源程序集为了保证信任正确的代码，就需要指定友元程序集的公钥。与很多文档叙述的相反，你需要的不是公钥令牌而是公钥本身。例如，思考一下如下的命令行和输出（为了显示更好的格式，稍微重新包装并修饰了一下），用于发现签名FriendAssembly.dll程序集的公钥：

```
c:\Users\Jon\Test>sn -Tp FriendAssembly.dll
Microsoft (R) .NET Framework Strong Name Utility Version 3.5.21022.8
Copyright (c) Microsoft Corporation. All rights reserved.
Public key is
0024000004800000940000000602000000240000525341310004000001
000100a51372c81ccfb8fba9c5fb84180c4129e50f0facdce932cf31fe
```

```
563d0fe3cb6b1d5129e28326060a3a539f287aaf59affc5aabc4d8f981
e1a82479ab795f410eab22e3266033c633400463ee7513378bb4ef41fc
0cae5fb03986d133677c82a865b278c48d99dc251201b9c43edd7bedef
d4b5306efd0dec7787ec6b664471c2

Public key token is 647b99330b7f792c
```

Source类的源代码现在需要将如下内容作为属性：

```
[assembly:InternalsVisibleTo("FriendAssembly,PublicKey=" +
"0024000004800000940000000602000000240000525341310004000001" +
"000100a51372c81ccfb8fba9c5fb84180c4129e50f0facdce932cf31fe" +
"563d0fe3cb6b1d5129e28326060a3a539f287aaf59affc5aabc4d8f981" +
"e1a82479ab795f410eab22e3266033c633400463ee7513378bb4ef41fc" +
"0cae5fb03986d133677c82a865b278c48d99dc251201b9c43edd7bedef" +
"d4b5306efd0dec7787ec6b664471c2")]
```

比较麻烦的是，你必须把公钥放在一行中，或者使用字符串连接符——在公钥中间留有空白会引起编译错误。如果我们真能指定令牌而非整个公钥的话，会让人非常高兴的，不过幸好这个丑陋的东西通常放置在AssemblyInfo.cs中，所以你不用经常去查看它。

理论上，有可能出现未签名的源程序集和签名的友元程序集。实际上，这没有多大用处，因为友元程序集通常想要一个指向源程序集的引用——你无法从一个签名的程序集中引用一个未签名的程序集。同样，签名的程序集也不能指定一个未签名的友元程序集，所以通常最终采取的方式是，如果其中任何一个程序集有签名，那么两个程序集都进行签名。

7.8　小结

现在，我们已经完成了C# 2新特性的旅程。我们在本章中研究的主题大致可以归为两类："值得引入"的改进可以让开发更流畅；"希望你不要使用的"特性，可以在你需要它们的情况下为你解决棘手的问题。如果把C# 2和对房子改进进行类比，那么我们之前章节讲到的主要特性堪比一个完整的房屋。我们在本章中看到的某些特性（比如分部类型和静态类）很像重新装饰了卧室，而类似命名空间别名这样的特性就好像为房屋安装了烟雾报警器——你也许永远都看不到它的好处，不过一旦你需要它们的时候，就知道它们存在的好处了。

C# 2中涉及很多特性——设计器解决了很多开发人员觉得痛苦的问题，而不是去完成包罗万象的目标。这也不是说这些特性不能一起很好地工作（例如，没有泛型的存在可空类型也就不会出现），不过也不存在一个需要用到每种特性的目标，除非你把综合的生产力算在内。

既然我们已经完成了C# 2的研究，下面该来研究C# 3了，它所带来的东西有很大的不同。

C# 3的几乎每个特性）都是LINQ这个宏伟蓝图的一个组成部分，这些技术的混合物可以很大程度上简化许多任务。

Part 3

第三部分

C# 3：革新写代码的方式

C# 2无疑是对C# 1的一次显著增强。在各种增强中，泛型尤其重要，它奠定了其他改变的基础（不仅仅是C# 2中，也包括C# 3）。但是，C# 2感觉有点儿像是各种特性的一个零碎的组合。请不要误解我的意思：它们确实组合得不错，但它们解决的是一系列独立的问题。在C#的开发阶段，这种组合方式是适用的，但C# 3就不一样了。

C# 3的几乎每个新特性都是为一种特定的技术——LINQ服务的。许多特性在LINQ的背景之外也非常有用，你绝对不应只在需要写一个查询表达式的时候才使用这些特性。但是，如果无法识别由本书接下来的5章中出现的"一套智力拼图零件"拼出的完整画面，那么同样是不可取的。

当我最初在2007年撰写C# 3和LINQ时，对其深邃的思想印象颇深。对该语言学习得越深入，你就越能清晰地感受到引入的不同元素之间的紧密联系。查询表达式十分优雅，特别是可以对进程内查询和LINQ to SQL这样的提供器使用相同的语法，这一点更是魅力四射。LINQ的前景可谓无限光明。

而在今天，我们可以回头看看LINQ究竟发展到了什么地步。在我的印象中，社区（特别是Stack Overflow）已经开始广泛使用LINQ，并且已经改变了我们处理面向数据任务的方式。数据库提供器也不再仅限于微软的产品——LINQ to NHibernate和SubSonic仅仅是众多选择中的两个。微软也一直没有停止过对LINQ的创新：第12章将介绍Parallel LINQ和Reactive Extensions，它们仍然使用熟悉的LINQ操作符，却以完全不同的方式处理数据。还有LINQ to Objects——最简单、最传统、最常规的LINQ提供器——已经在行业内 得到了最普遍的应用。那种编写不同的用于过滤的循环、用于查找最大值的代码、检查集合中的每一项是否满足条件的日子，已经一去不复返了。

尽管LINQ已经被广泛采用，但从我收集到的一些提问中不难发现，很多开发者仍然将LINQ看成是一个魔术盒子。查询表达式与直接使用扩展方法相比有什么不同？什么时候真正读取数据？如何能让它更高效地工作？尽管你可以通过实践和阅读博客来学习LINQ，但如果你看到了它在语言级别是如何工作的，然后学习各种不同的库，将对你水平的提高产生莫大的帮助。

这不是一本介绍LINQ的书，因此我们仍然会将注意力集中在语言特性层面，而不会深入Entity框架并发性之类的细节。但如果你掌握了单独的语言元素，并了解了它们如何协同，就会很容易理解那些特定的提供器的细节。

第 8 章

用智能的编译器来防错

本章内容

- 自动实现的属性
- 隐式类型的局部变量
- 对象和集合初始化程序
- 隐式类型的数组
- 匿名类型

首先，让我们以结束C# 2的方式来开始C# 3——学习相对简单的各项特性。但是，在通向LINQ的路上，它们只是最初的几个小台阶而已。每一项特性都可脱离LINQ使用。为了使LINQ能够有效地运用，代码首先必须简化到它要求的程度，这些特性对于简化代码都是相当重要的。

要注意的一个重点在于，虽然C# 2最重要的两个特性（泛型和可空类型）要求对CLR进行改动，但在随.NET 3.5提供的CLR中，却并没有发生重大的变化。有的只是一些bug修正，基本的东西没有变。框架库进行了扩充以支持LINQ，基类库也引入了几个新特性，但那是另一回事。你有必要弄清楚哪些改变只是C#语言的改变，哪些是库的改变，以及哪些是CLR的改变。

这意味着C# 3中的所有新特性都是因为编译器现在能帮你做更多的事情。在本书第二部分中就已经看到了这样的情形（如匿名方法和迭代器块），C# 3继续走着相同的路线。本章将讨论C# 3的以下新特性：

- **自动实现的属性**——编写由字段直接支持的简单属性，不再显得臃肿不堪；
- **隐式类型的局部变量**——根据初始值推断变量的类型，从而简化局部变量的声明；
- **对象和集合初始化程序**——用一个表达式就能轻松创建和初始化对象；
- **隐式类型的数组**——根据内容推断数组的类型，从而简化数组的创建表达式；
- **匿名类型**——允许你创建新的“临时”类型来包含简单的属性。

在描述新特性能做什么的同时，我还会就其用法提出一些建议。C# 3的许多特性都要求开发者有一定的审慎和克制。这并不是说这些特性的功能不强，用处不大（事实刚好相反），只是说我们的大前提仍然是保证代码清晰和易读。使用最新和最强大的语法糖时，必须以不违反这个前提为准。

本章（以及本书剩余部分）提出的一些见解很少有非常绝对的。可能在旁观者看来，可

读性增加了。随着你越来越熟悉新的特性，它们的可读性在你的眼中就会变得越来越好。但要强调的是，除非你有很好的理由认定自己是唯一看自己代码的人，否则就应该照顾一下你的同事的习惯。

好了，纸上谈兵差不多够了。现在让我们从一个不应引起任何争论的特性开始。简单而高效的“自动实现的属性”使我们的编码变得更简单了。

8.1 自动实现的属性

我们的第一个特性或许是C# 3中最简单的。事实上，它甚至比C# 2引入的任何新特性都要简单。虽然如此（或者可能正因为如此），它在许多情况下能“拿起来就用”。当你阅读第6章有关迭代器块的内容时，可能并不能马上想到能用它对自己当前的代码库进行什么改进。但是，却很少有不能用自动实现的属性进行修改的不平凡的C# 2程序。这个异常简单的特性可使你使用比以前少的代码来表示普通属性。

何谓“普通属性”？我的意思是指可读/可写并将值存储到一个非常直观的私有变量中的属性，不做任何校验，也没有其他自定义代码。这些属性只是占用了几行代码，但与它们所表达的简单的概念相比，还是显得过多了。C# 3执行了一个简单的编译时转换，减少了这种繁文缛节，如图8-1所示。

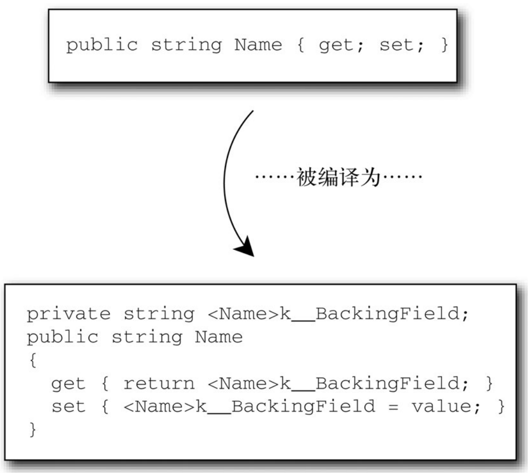


图8-1 对自动实现属性的转换

当然图8-1中最下面的代码不是非常有效的C#。该字段使用了不友好的名称来防止命名冲突，这种方式与匿名方法和迭代器块相同。它是上面自动实现的属性所生成的有效的代码。

以前为了简单，你可能经常使用公共变量，但现在没有理由不使用属性了。对于一次性代码来说更是这样，因为我们都知道这种代码的生存期远比我们想象的要长。

说明　术语："自动属性"与"自动实现的属性"　当人们最早开始讨论"自动实现的属性"时（早在完整的C# 3规范发布之前），它们被称为"自动属性"。我个人觉得它比全名要简练得多，而且在社区也应用得更广。这里不会引起歧义，因此在本书剩余的部分，我会把"自动属性"和"自动实现的属性"当作同义词来使用。

C# 2允许为取值方法和赋值方法指定不同的访问权限。这个特性并未取消，现在仍可使用。另外，还可以创建静态的自动属性。然而，静态自动属性基本没什么意义。虽然大多数类型都不会声称自己有线程安全的实例成员，但公开可见的静态成员通常应该是线程安全的，编译器在这方面帮不上你任何忙。代码清单8-1展示了一个安全却没什么用处的静态自动属性，用来计算创建了多少个该类的实例，同时还包含两个实例属性，表示一个人的名字和年龄。

代码清单8-1　统计创建了多少个实例的Person类

```
public class Person
{
    public string Name { get; private set; }                    // 声明有公有取值方法的属性
    public int Age { get; private set; }

    private static int InstanceCounter { get; set; }            // 声明私有的静态属性和锁
    private static readonly object counterLock = new object();

    public InstanceCountingPerson(string name, int age)
    {
        Name = name;
        Age = age;

        lock (counterLock)                                      // 在访问静态属性时使用锁
        {
            InstanceCounter++;
        }
    }
}
```

在代码清单8-1中，我们使用锁来确保不会产生线程问题，并且只要访问该属性，就需要使用相同的锁。使用`Interlocked`类应该是更好的选择，但它要求访问字段。总之，我见过的使用静态自动属性的唯一情况是，取值方法是公共的，赋值方法是私有的，并且赋值方法只能在类型初始化程序中使用。

代码清单8-1中的其他属性代表的是人的名字（Name）和年龄（Age），这告诉了我们一个令人振奋的消息：使用自动属性根本不需要任何思考。对于在C#之前版本中已经实现过的那些属性来说，不使用自动属性也没有什么好处①。

写自己的结构时，如果使用自动属性，那么会有一个小问题：所有构造函数都需要显式地调

① 这当然是对只读/只写属性来说的。如果要创建一个只读属性，你可以选择使用一个只读后备字段以及一个属性，该属性带有一个`getter`方法来返回它。这使你不会意外地在该类中写入这个属性，而这种意外写入的情况对"公有读、公有写"自动属性来说是有可能发生的。

用一下无参构造函数this()，只有这样，编译器才知道所有字段都被明确赋值了。由于字段是匿名的，所以不能直接设置它们。同时，在所有字段都被设置之前，也不能使用这些属性。如果要使用属性，唯一的办法就是调用无参构造函数，它会将字段设成它们的默认值。比如，如果想写一个有单个整数属性的结构，将是无效的。

```
public struct Foo
{
   public int Value { get; private set; }

   public Foo(int value)
   {
      this.Value = value;
   }
}
```

但如果像下面这样显式地调用无参构造函数，就完全可以了。

```
public struct Foo
{
   public int Value { get; private set; }

   public Foo(int value) : this()
   {
      this.Value = value;
   }
}
```


关于自动实现的属性，以上便是你需要了解的全部内容。但它们也不是完美无缺的——例如，没有办法在声明它们的时候指定初始的默认值，也没有办法把它们变成真正的只读属性（使用私有赋值方法，是最方便的只读属性的解决方案）。

假如C# 3的所有特性都像这么简单，那么全部内容用一章的篇幅即可讲完。虽然实情并非如此，但仍有部分特性不需要花费太多的篇幅来解释。我们的下一个主题是如何在另一个常见但又特定的情况下（声明局部变量）移除重复的代码。

8.2　隐式类型的局部变量

第2章讨论了C# 1类型系统的本质。我特别指出，C# 1的类型系统是静态、显式和安全的。这个结论在C# 2中同样成立。在C# 3中，它仍然几乎是成立的。“静态”和“安全”仍然是成立的（和第2章一样，要忽略显式的不安全代码）。另外，多数时候，它仍然是显式类型的，只是可以要求编译器帮你推断局部变量的类型[①]。

8.2.1　用**var**声明局部变量

如果要使用隐式类型，唯一要做的就是将普通局部变量声明中的类型名称替换为var。虽然

① C# 4再次改变了游戏规则，允许你使用动态类型，我们将在第14章进行介绍。不过，截至目前（包括C# 3）C#仍然是完全静态类型的语言。

有一些限制（稍后会详述），但基本上就是将

```
MyType variableName = someInitialValue;
```

替换成：

```
var variableName = someInitialValue;
```

只要someInitialValue的类型是MyType，两行代码的结果（也就是最终的编译结果）就是完全一样的。编译器所做的工作很简单，就是获取初始化表达式在编译时的类型，并使变量也具有那种类型。类型可以是任何普通的.NET类型，包括泛型、委托和接口。变量仍然是静态类型的，只是你在代码中没有写类型的名称而已。

理解这一点是很重要的，因为许多开发者在刚开始接触这个特性时都会感到恐惧：var使C#变成了动态类型或者弱类型的语言。实际上，这种理解是完全错误的。要解释这一点，最好的办法就是演示一些无效的代码：

```
var stringVariable = "Hello, world.";
stringVariable = 0;
```

上述代码是无法编译的，因为stringVariable的类型是System.String，而你不能将值0赋给一个字符串变量。在许多动态语言中，上述代码是可以编译的，造成在编译器、IDE或运行时环境的眼中，变量的类型变得无关紧要。使用var和使用COM或VB6中的VARIANT类型不一样。变量是静态类型的，只是类型要由编译器推断。我要对我的唠叨抱歉，但这是相当重要的一点，经常引起混乱。

在Visual Studio中，将鼠标对准声明中的var部分（如图8-2所示），就可以看到编译器为变量推断的类型是什么。注意，泛型Dictionary类型的类型参数也同样进行了推断。这对你来说应该不显得陌生，因为这正是显式声明局部变量时的行为。

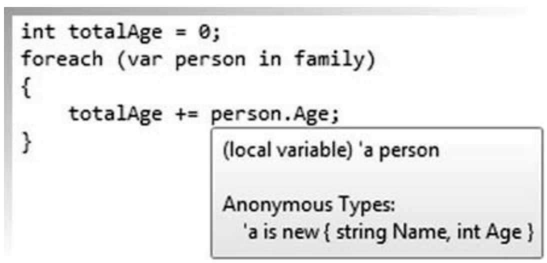


图8-2　在Visual Studio中将鼠标对准var，会显示所声明变量的类型

这种工具提示（tooltip）不仅仅是在声明时可用。正如你可能期望的那样，变量名以后在代码中出现时，将鼠标对准它，也会有一个工具提示显示变量的类型。图8-3对此进行了演示，此处使用了相同的声明，然后我将鼠标对准了use变量的一处使用。这同样是和普通局部变量一样的行为。

有两个原因促使我们在这里使用Visual Studio。第一个是它很好地证明了现在使用的仍然是静态类型——编译器清楚地知道变量的类型。第二个是可以很容易地知道真实类型是什么，即使

是在一个方法的深处。稍后讨论隐式类型的优缺点时，这会成为很重要的一点[①]。但是，我想先指出它的一些限制。

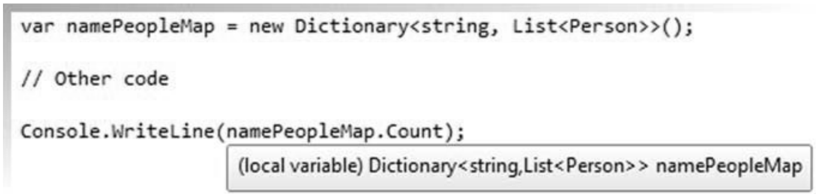


图8-3　鼠标对准一个使用隐式类型的局部变量的地方，会显示它的类型

8.2.2　隐式类型的限制

不是在所有情况下都能为所有变量使用隐式类型，只有在以下情况下才能用它：

- 被声明的变量是一个局部变量，而不是静态字段和实例字段；
- 变量在声明的同时被初始化；
- 初始化表达式不是方法组，也不是匿名函数[②]（如果不进行强制类型转换）；
- 初始化表达式不是`null`；
- 语句中只声明了一个变量；
- 你希望变量拥有的类型是初始化表达式的编译时类型；
- 初始化表达式不包含正在声明的变量[③]。

第3点和第4点比较有趣。你不能像下面这样写：

```
var starter = delegate() { Console.WriteLine(); };
```

因为编译器不知道使用什么类型。但可以像这样写：

```
var starter = (ThreadStart) delegate() { Console.WriteLine(); };
```

但是，如果这样写，最好一开始就显式声明变量。同样的道理也适用于`null`的情况——可以对`null`进行恰当的强制类型转换，但那样做就没有意义了。

注意，可以将方法调用的结果或属性作为初始化表达式使用——并非只能使用常量或构造函数调用。例如，你可以使用：

```
var args = Environment.GetCommandLineArgs();
```

在这种情况下，`args`将具有`string[]`类型。事实上，用方法调用的结果来初始化一个变量，可能是隐式类型最常见的一个应用，如同在LINQ中那样。我们以后会看到具体的例子，看完多个例子以后，你会记住这里说过的话。

① 作者的意思是说，如果不使用Visual Studio，有时就不好判断类型是什么，而这正是隐式类型的一个缺点。
——译者注

② “匿名函数”这个术语同时包括了匿名方法和Lambda表达式，后者将在第9章详述。

③ 这种情况十分少见，但在普通的声明中也不是没有可能。

另外值得注意的是，可以为using、for或foreach语句的开头部分中声明的局部变量使用隐式类型。例如，下面这些都是合法的（当然，要有适当的语句块）：

```
for (var i = 0; i < 10; i++)
using (var x = File.OpenText("test.dat"))
foreach (var s in Environment.GetCommandLineArgs())
```

声明的3个变量将分别具有int、StreamReader和string类型。

当然，允许这样做，并不意味着就应该这样做。下面来看看促使和阻止我们使用隐式类型的原因。

8.2.3　隐式类型的优缺点

“什么时候适合使用隐式类型”这个问题在社区中很容易引起激烈争论。人们的观点不一，有最极端的“在任何地方都用”和“在任何地方都不用”，也有介于两者之间的一些说法。8.5节会讲到，为了使用C# 3的另一个特性（匿名类型），你经常*都要*使用隐式类型。当然，也可以完全避免使用匿名类型，但那样就是因噎废食了。

之所以使用隐式类型（暂不考虑匿名类型的问题），主要原因是它不仅减少了代码的输入量，还减少了屏幕上显示的代码量（这就意味着可读性增强）。尤其是在涉及泛型时，类型名称可能变得相当长。图8-1和图8-2使用了一个叫做Dictionary<string,List<Person>>的类型。这个类型名称总共含有33个字符。如果它在一行上出现两次（一次是声明，一次是初始化），那么最后得到的将是一行非常长的代码，而这仅仅是为了声明和初始化一个变量！一种解决方法是使用别名，但那样就人为地拉长了“真实”类型与使用它的代码之间的距离（至少在概念上如此）。

读取代码时，相同的长类型名称在同一行上没必要出现两次——如果它们显然*应该*是同一个类型的话。如果在屏幕上看不见声明，是否使用了隐式类型就没有什么区别（用于查看变量类型的所有方法仍然有效）。如果在屏幕上看得见声明，那么用于初始化变量的表达式无论怎样都能告诉你类型是什么。

此外，使用var还改变了代码的重心。有时，你希望读者将注意力放到确切的类型上，因为它们更重要。例如，尽管泛型的SortedList和SortedDictionary类型具有类似的API，但他们的性能则完全不同，这对于特定代码片段来说可能是很重要的。而有时，你真正关心的是正在执行的操作：只要能达到相同的目标，你不会在意用于初始化变量的表达式是否发生了变化[①]。使用var可以将读者的注意力从变量的声明转移到使用上——是代码*做了什么*，而不是*怎么做*。

所有这些听起来都很不错，那么有什么理由*阻止*我们使用隐式类型呢？可读性是最重要的一个原因！这听起来似乎有点儿自相矛盾，因为我们刚刚还说可读性是隐式类型的优势之一。我们现在的意思是，假如不显式地指出要声明的变量是什么类型，只通过读代码的方式，可能不好确定它的类型。这打破了我们“先声明，再赋值”的这种将声明和初始化相分离的思维习惯。至于打破到什么程度，则取决于读代码的人和具体的的初始化表达式。

---

① 我知道这听上去有点像鸭子类型：“只要还能嘎嘎叫，我是鸭子我骄傲。”但不同之处在于，我们仍然是在编译时检查它是否能嘎嘎叫，而不是执行时。

如果显式调用一个构造函数，那么应该能轻松判断创建的是什么类型。如果调用一个方法或使用一个属性，就要取决于能否通过查看调用来轻松地判断返回类型。有的时候，也许并不容易猜准编译器推断的类型。整数常量就是这样的一个例子。对于下面声明的每一个变量，你能以多快的速度判断出它们的类型？

```
var a = 2147483647;
var b = 2147483648;
var c = 4294967295;
var d = 4294967296;
var e = 9223372036854775807;
var f = 9223372036854775808;
```

答案依次是：`int`、`uint`、`uint`、`long`、`long`和`ulong`——具体使用什么类型要取决于表达式的值。从处理常量的方式上说，这里没有什么新东西，C#的行为自始至终都是这样的。但是，在这种情况下使用隐式类型，很容易就会写出让人“雾里看花”的代码。

很少有人明确地指出这一点，但我认为在针对隐式类型产生的诸多忧虑的背后，隐藏着一个简单的论点：“它感觉就是不太对！”如果你用C风格的一种语言写了多年的程序，在面对C#时，会紧张不安，无论如何告知自己语言在幕后仍然是静态类型的，你依然会觉得不安。听起来似乎不太理性，却是真实存在的。如果感觉不舒服，自然会影响到写代码的效率。如果你觉得使用隐式类型利大于弊，那么不妨用它。这与你的性格有关，你可能会尝试强迫自己去接受并熟悉隐式类型——但这不是必须的。

8.2.4　建议

下面根据我自己使用隐式类型的经验提出了一些建议。记住，这些仅仅是建议，你完全可以不照办。

- 如果让读代码的人一眼就能看出变量的类型是很重要的，就使用显式类型。
- 如果变量直接用一个构造函数初始化，而且类型名称很长（用泛型时经常会这样），就考虑使用隐式类型。
- 如果变量的确切类型不重要，而且它的本质在当前上下文中已很清楚，就用隐式类型，从而不去强调代码具体是如何达到其目标的，而是关注它想要达到什么目标这个更高级别的主题。
- 在开发一个新项目的时候，与团队成员就这件事情进行商议。
- 如有疑虑，一行代码用两种方式都写一写，根据直觉选一个最“顺眼”的。

除非能从简化代码获益巨大，否则在生产代码中我更愿意使用显式类型。不过，隐式类型很适合一次性代码和测试代码。坦白说，我现在觉得挺矛盾的。我在写代码时更愿意使用显式类型，只为了能更简单一点，即便涉及的类型名称并没有那么繁杂。即使编程风格某些方面的一致性相当重要，我也没发现这种不协调会引起什么问题。

总之，我的建议是不要因为“它是一个新东西”，或者仅仅是因为自己想偷懒，想少打几个字，就选择使用隐式类型。相反，如果它能使代码更整洁，使你能将注意力集中在代码更重要的

元素上，就使用它。本书剩余部分会广泛地使用隐式类型，理由很简单，在书上印刷代码要比在屏幕上显示难一些，因为书的宽度不够。

以后讨论匿名类型时，还会回到隐式类型的主题上。这是因为在某些情况下，你必须强迫编译器推断一些变量的类型。在此之前，让我们先来看一看C# 3如何在一个表达式中更简单地构造和填充一个新对象。

8.3　简化的初始化

有的人以为面向对象的语言早就理顺了对象的创建过程。毕竟，一个对象在使用之前，必须先由某个东西创建它，不管是直接通过你的代码，还是通过某种形式的工厂方法。在C# 2中，很少有什么语言特性是专门为了简化初始化而设计的。如果使用构造函数的实参还不能满足你的需求，那么很不幸——只能创建对象，然后用属性设置或类似的方式来手动初始化它。

如果想一次为数组或其他集合创建多个对象，这种设计尤其让人恼火。你无法使用单一的表达式来初始化对象，只能使用局部变量来进行临时的操作，或创建辅助方法，根据参数来执行相应的初始化。

如本节马上要讲到的，C# 3可以通过各种方式来拯救你于水火之中。

8.3.1　定义示例类型

本节要使用的表达式称为*对象初始化程序*（object initializers）。初始化列表指定了在对象创建好之后，如何对其进行初始化。你可以设置属性，设置属性的属性（不用担心，实际会比你想象的简单），还可以向通过属性访问的集合中添加内容。为了对此进行演示，我们将再次使用Person类。前面的名称和年龄仍将保留，它们是作为可写属性给出的。我们将提供一个无参数的构造函数，以及一个使用name作为参数的构造函数。我们还要添加一个朋友列表和一个人的家庭住址。它们是只读属性，但通过对获取的对象进行处理，仍可对其进行修改。在一个简单的Location类中，我们提供了Country（国家）和Town（城市）属性来表示一个人的家庭住址。代码清单8-2展示了类的完整源代码。

代码清单8-2　一个相当简单的Person类，用于进一步演示

```
public class Person
{
   public int Age { get; set; }
   public string Name { get; set; }

   List<Person> friends = new List<Person>();
   public List<Person> Friends { get { return friends; } }

   Location home = new Location();
   public Location Home { get { return home; } }

   public Person() { }

   public Person(string name)
```

```
    {
       Name = name;
    }
}
public class Location
{
   public string Country { get; set; }
   public string Town { get; set; }
}
```

代码清单8-2很容易就看得懂，但要注意的是，无论朋友列表还是家庭位置，都是在Person对象创建时，以一种“留空”的方式来创建的，而不是仅仅保留空引用。并且，它们还是只读的。这一点的重要性以后才会显现出来，但就目前来说，先让我们将注意力集中在代表一个人的名称和年龄的属性上。

8.3.2 设置简单属性

既然Person类型已经定义好了，我们想利用C# 3的新特性来创建它的一些实例。在本节中，我们要研究如何设置Name和Age属性，其他属性以后再讨论。

事实上，对象初始化程序最常用于设置属性，但这里描述的所有语法糖同时还适用于字段。但是，你大多数时候使用的都将是属性。而在一个封装良好的系统中，是不大可能访问到字段的，除非要在一个类型自己的代码中创建该类型的一个实例。当然，你是可以使用字段的。所以当后文说到“属性”时，不妨自行将它们读作“属性和字段”。

8

进行了一番说明之后，让我们进入主题。假定现在要创建一个名为Tom的人，他的年龄是9岁。在C# 3之前，可以采取两种方式：

```
Person tom1 = new Person();
tom1.Name = "Tom";
tom1.Age = 9;

Person tom2 = new Person("Tom");
tom2.Age = 9;
```

第一种方式直接使用无参构造函数，然后设置这两个属性。第二种方式则使用了一个能设置名字的重载的构造函数，然后再设置年龄。当然，这两种方法在C# 3中仍然可以使用，但现在多了另外三种方案：

```
Person tom3 = new Person() { Name = "Tom", Age = 9 };
Person tom4 = new Person { Name = "Tom", Age = 9 };
Person tom5 = new Person("Tom") { Age = 9 };
```

在每一行末尾，用大括号括的就是对象初始化程序。同样，这是编译器耍的一个花招。用于初始化tom3和tom4的IL是完全一样的，并且与初始化tom1的IL也几乎[①]完全一样。可以预见，tom5的IL和tom2的IL也几乎完全一样。注意在声明tom4时，我们省略了构造函数的圆括号。如果类型有一个无参的构造函数，就可以使用这种简写方式，这正是在编译完的代码中调用的。

① 事实上，变量的新值是在设置完所有属性之后才被分配的。在那之前，会使用一个临时的局部变量。这并不十分重要，但如果你偶然通过初始化器中断进入了调试器，了解这一点可以避免混淆。

调用了构造函数之后，接着就要设置指定的属性了。它们根据在对象初始化程序中出现的顺序来设置。任何特定的属性最多只能指定一次，例如，不能设置两次Name属性。（但是，你可以先调用以名称作为参数的那个构造函数，再设置Name属性。编译器不会阻止你这样做，虽然这样做是没有意义的。）作为属性值使用的表达式可以是任何表达式，只要它本身不代表一个赋值——你可以调用方法、创建新对象（可以使用另一个对象初始化程序），等等。

你也许觉得这个设计没什么用——虽然节省了一两行代码，但那不应该成为将语言变得更复杂的理由吧？但是，这里有一个很容易被忽视的地方：我们不仅仅是用一行代码来创建了一个对象——我们实际是用一个表达式创建了它。这个差异有时会变得非常重要。

假定你想创建Person[]类型的一个数组，其中有一些预定义的数据。即使不使用我们以后会讲到的隐式数组类型，代码也是非常简洁和易读的：

```
Person[] family = new Person[]
{
   new Person { Name = "Holly", Age = 36 },
   new Person { Name = "Jon", Age = 36 },
   new Person { Name = "Tom", Age = 9 },
   new Person { Name = "William", Age = 6 },
   new Person { Name = "Robin", Age = 6 }
};
```

在像这样一个简单的例子中，我们可以写一个构造函数来同时获取名字和年龄作为参数，并采用与C# 1和C# 2类似的方式来初始化数组。然而，合适的构造函数并非总是摆在那儿供你使用的。并且，如果构造函数同时要获取几个参数，那么经常都会分不清每个参数代表什么意思。当一个构造函数需要获取5个或6个参数时，我发现我更需要依赖“智能感知”技术。在这种情况下，使用恰当的属性名可极大地增强可读性[①]。

这种形式的对象初始化程序或许是你最常用的。然而，还有另外两种形式：一种用于设置子属性；另一种用于添加到集合。让我们先来讨论子属性——属性的属性。

8.3.3　为嵌入对象设置属性

到现在为止我们发现，Name和Age属性是很容易设置的。但是，Home属性不能这样设置，因为它是只读的。然而，我们可以先获取Home属性，并对它（location）的属性进行设置，以这种方式来设置一个人的城市和国家。在语言规范中，这称为“设置一个嵌入对象的属性”。

为了把话说清楚，请看以下C# 1代码：

```
Person tom = new Person("Tom");
tom.Age = 9;
tom.Home.Country = "UK";
tom.Home.Town = "Reading";
```

填充家庭位置时，每个语句都是先执行一个取操作来取得Location实例，再对那个实例中的对应的属性执行一次赋值操作。看起来似乎没什么新东西，但应该认真仔细看，否则很容易错

① C# 4在这里提供了一种替代方法：命名实参，我们将在第13章进行介绍。

过幕后发生的事情。

C# 3可以在一个表达式中完成这些工作，如下例所示：

```
Person tom = new Person("Tom")
{
   Age = 9,
   Home = { Country = "UK", Town = "Reading" }
};
```

编译后的代码和刚才展示的片段代码是完全一样的。编译器发现等号右侧的是另一个对象初始化程序，所以会适当地将属性应用到嵌入对象。

在初始化Home的部分中，没有出现new关键字，这是值得注意的。要想知道新对象是在什么位置创建的，以及对已有对象来说，对象的属性又是在什么位置设置的，只需查看new在初始化列表中的什么位置出现就可以了。每当一个新对象被创建时，new关键字就会在某个地方出现。

> **说明** **关于对象初始化程序的编码格式** 和几乎所有C#特性一样，空白是影响不了语义的。如果愿意，完全可以删除对象初始化程序中的所有空白，把它们全部放到一行中。在“长的代码行”和“多的代码行”之间，存在着一个最佳平衡点，找到这个平衡点完全是你自己的事。

我们已经处理好Home属性，但Tom的朋友们怎么办呢？List<Person>有一些属性是可以设置的，但这些属性都不能在列表中添加项。这就该下一个C# 3新语言特性——集合初始化列表出场了。

8

8.3.4 集合初始化程序

创建一个带有一些初始值的集合，这是相当常见的一个任务。在C# 3之前，能为此提供一点帮助的唯一语言特性就是数组创建，而且在许多时候，即使那样都还是显得非常笨拙。C# 3新增了集合初始化程序（collection initializer），利用它你使用和数组初始化程序一样的语法，但支持任意集合，而且更灵活。

1. 使用集合初始化程序来创建新集合

作为第一个例子，我们使用现在已经非常熟悉的List<T>类型。如果要在C# 2中填充列表，要么是传入一个现有的集合，要么是先创建空列表，再重复调用Add。C# 3的集合初始化列表采用的是后一种方法。

假定现在要在一个字符串列表中填充一些名字——以下是C# 2代码（左列）和等价的C# 3代码（右列）。

```
List<string> names = new List<string>();
names.Add("Holly");
names.Add("Jon");
names.Add("Tom");
names.Add("Robin");
names.Add("William");
```

```
var names = new List<string>
{
   "Holly", "Jon", "Tom",
   "Robin", "William"
};
```

和使用对象初始化程序一样，如果愿意，你可以指定构造函数参数，或者显式/隐式地使用无参构造函数。这里之所以使用隐式类型，部分原因就是为了节省篇幅——其实显式声明names变量同样是很好的一种做法。除了减少代码行数（在不影响可读性的前提下），集合初始化列表还有另外两个更大的好处：

- "创建和初始化"部分被算作一个表达式；
- 代码整洁了许多。

将集合作为传给一个方法的实参，或者作为一个更大的集合中的元素使用时，第一点就会变得相当重要。这种情况相对来说较少发生（虽然足够使其有用）。在我看来，第二点才是使"集合初始化列表"成为C# 3的"重磅级"新特性的真正原因。看一下右侧的代码，你会很容易地发现所有需要的信息，而且每样信息都只写了一次。变量名只出现了一次，要使用的类型只出现了一次，而且被初始化的集合的每一个元素都只出现一次。一切都相当简单，比C# 2代码清楚得多。在C# 2的代码中，真正有用的信息往往被淹没在大量无价值的信息中。

集合初始化列表并非只能应用于列表。任何实现了IEnumerable的类型，只要它为初始化列表中出现的每个元素都提供了一个恰当的公有的Add方法，就可以使用这个特性。Add方法可以接受多个参数，只需把值放到另一对大括号中。最常见的应用就是创建字典，例如，假定你要创建一个将名字映射到年龄的字典，可以使用以下代码：

```
Dictionary<string,int> nameAgeMap = new Dictionary<string,int>
{
   { "Holly", 36 },
   { "Jon", 36 },
   { "Tom", 9 }
};
```

在这个例子中，Add(string, int)方法会被调用3次。如果Add有多个重载版本，那么初始化列表中的每个不同的元素都可以调用不同的重载（版本）。如果不能为某个具体的元素找到兼容的重载（版本），代码就无法通过编译。设计者在这里有两个非常有趣的决策：

- 类型虽然必须实现IEnumerable，但这个事实永远不会被编译器利用；
- 只要按名字能找到Add方法就可以了，不要求指定任何接口。

这两个决策都是从实用的角度出发的。要求实现IEnumerable是合理的，这是为了检查类型是否真的是某种形式的集合，而允许使用Add方法的任何公有重载版本（而不是要求一个固定的签名）可以实现简单的初始化（就像前面的字典例子那样）。

但在C# 3语言规范的一份早期的草案中，却要求必须实现ICollection<T>，而且只会调用单参数的Add方法（由接口指定），而不允许不同版本的重载。这使得语言看起来更"纯净"，但实现IEnumerable的类型要比实现ICollection<T>的多得多，而且使用单参数的Add方法显得十分不便。例如在前面的例子中，就必须为初始化程序中的每个元素都显式创建一个KeyValuePair<string,int>的实例。牺牲一点点带有学术气息的"纯净"，换来了语言巨大的实用性！

2．在其他对象初始化程序中填充集合

前面只演示了如何独立地使用集合初始化程序来创建一个全新的集合。它们还可与对象初始

化程序组合使用，来填充嵌入的集合。回到Person类的例子。Friends属性是只读的，所以不能创建一个新集合，再将其指定给朋友属性。但是，我们可以向属性getter返回的任何集合中添加内容。具体做法和从前为嵌入对象设置属性的语法相似，只是这里要指定的是一个集合初始化程序，而不是指定一系列属性。

下面为Tom创建了另一个Person实例，来对此进行演示，这一次添加了他的朋友（代码清单8-3）。

代码清单8-3　使用对象和集合初始化程序来构建一个“富对象”

```
Person tom = new Person                                   <-- 调用无参数构造函数
{
    Name = "Tom",                                         <-- 直接设置属性
    Age = 9,
    Home = { Town = "Reading", Country = "UK" },          <-- 初始化嵌入对象
    Friends =
    {
        new Person { Name = "Alberto" },                  <-- 用更进一步的对象初始化程序来初始化集合
        new Person("Max"),
        new Person { Name = "Zak", Age = 7 },
        new Person("Ben"),
        new Person("Alice"),
        {
            Age = 9,
            Home = { Town = "Twyford", Country = "UK" }
        }
    }
};
```

代码清单8-3使用了我们迄今为止介绍过的对象和集合初始化程序的所有特性。主要的看点就是集合初始化程序，它本身在内部又使用了各种不同形式的对象初始化程序。注意，我们不是新建一个集合，而是向现有集合中添加内容。（如果属性包含赋值方法，我们仍然可以使用集合初始化程序语法来创建一个新的集合。）

甚至可以更进一步，指定朋友的朋友、朋友的朋友的朋友，等等。不能用这个语法做的就是指定Tom是Alberto的朋友——你无法访问正在初始化的对象，因此不能表示循环关系。这偶尔会造成不便，但一般不会成为问题。

在对象初始化程序中对集合进行初始化，有点儿像是“独立集合初始化列表”和“设置嵌入对象的属性”之间的一个交叉。针对集合初始化程序中的每一个元素，都会调用集合属性（本例是Friends）的取值方法，然后为返回值调用一个恰当的Add方法。在添加元素之前，不会采取任何方式先清除集合中的内容。例如，假如你以后决定某人应该始终是他/她自己的朋友，并在Person构造函数中将this添加到朋友列表中，那么使用集合初始化列表只会添加（除他/她自己之外）额外的朋友。

如你所见，集合和对象初始化程序的组合，可用于填充整个对象树。但在什么时候以及在什么地方，这才会真正地发生呢？

8.3.5　初始化特性的应用

试着指出这些特性在什么情况下有用，这感觉有点儿像是打地鼠——每当你觉得考虑到了全部的情况时，另一个例子又出现了。我这里只演示3个例子，它们的作用是抛砖引玉，希望你能联想出其他的应用。

1.“常量”集合

我经常需要用到一种本质上是“常量”的集合（通常是一个映射）。当然，从C#语言的角度看，它不可能成为一个常量，但它可以声明为静态和只读类型，并会醒目地警告你不应该更改它。（它通常是私有的，所以没有问题。此外还可以使用`Read-OnlyCollection<T>`。）为了填充映射，通常需要编写静态构造函数或辅助方法。现在，使用C# 3的集合初始化程序，可以轻松设置所有内联的东西。

2. 设置单元测试

写单元测试时，我经常都需要只为一个测试填充一个对象，通常把对象作为参数传给我当时要测试的方法，这对实体类来说尤其常见。用普通的方式来写所有初始化代码，会显得非常繁琐，而且会使代码的读者看不清楚对象的基本结构，就类似于在文本编辑器中打开一个XML文档时，XML创建代码经常会干扰到整篇文档的顺利阅读。通过对象初始化程序的恰当缩排，嵌套结构的对象层次可通过代码结构清楚地展现出来，而且会使值变得更加醒目。

3. `builder`模式

由于种种原因，你有时需要为单个方法或构造函数调用指定多个值（参数）。在我看来，最常见的情况莫过于创建不易变对象。与其使用庞大的参数列表（由于每个实参的含义不够明确，会导致很严重的可读性问题[1]），不如使用`builder`模式：使用适当的属性创建一个易变类型，然后将`builder`的实例传递给构造函数或方法。框架中的`ProcessStartInfo`类型就是一个很好的例子——设计者们完全可以用不同的参数集来重载`Process.Start`，但使用`ProcessStartInfo`可以使一切变得清晰。

我们可以使用对象和集合初始化程序以一种清晰的方式来创建`builder`对象——如果你愿意，甚至可以在调用原始成员时，将其制定为内联的。当然，首先你还是应该编写一个`builder`类型，自动属性会对你有所帮助。

4. <在此插入你喜爱的应用>

当然，在你平时接触到的代码中，肯定还能找到这些新特性的其他使用场景，对于这些应用我是绝对鼓励的。几乎没有不使用它们的理由，除非是可能会使还不熟悉C# 3的开发者感到迷惑。用一个对象初始化程序仅仅来设置一个属性（而不是用一个单独的语句来显式地设置它），你可能觉得这样做太夸张了。我只能说那是出于“审美”方面的考虑，在这方面，我无法给你提供太多客观的指导。和显式类型一样，最好的办法是两种方式都试试，看看你自己（和你的团队）在读代码时的喜好。

---

① 诚然，C# 4的命名实参会改善这一问题。

到目前为止，我们已经探讨了很多不同的特性：简便地实现属性，简化局部变量声明，以及在单一的表达式中填充对象。在本章剩余部分，我们将逐渐将这些主题融合到一起，使用更多的隐式类型和更多的对象填充，并在不提供任何实现细节的前提下创建完整的类型。

我们的下一个主题是“隐式类型的数组”。在阅读使用了这个特性的代码时，你会发现它和集合初始化列表颇为相似。以前说过，数组的初始化在C# 1和C# 2中显得有点笨拙。如果我说它在C# 3中已经被理顺了，相信你一点也不会感到惊讶。让我们看一下这个特性。

8.4 隐式类型的数组

在C# 1和C# 2中，作为变量声明和初始化的一部分，初始化数组的语句是相当整洁的。如果想在其他地方声明并初始化数组，就必须指定具体数组类型。例如，以下语句编译起来没有任何问题：

```
string[] names = {"Holly", "Jon", "Tom", "Robin", "William"};
```

但这种写法不适用于参数。假定要调用MyMethod方法，该方法被声明为void MyMethod (string[] names)，那么以下代码是无法编译的：

```
MyMethod({"Holly", "Jon", "Tom", "Robin", "William"});
```

相反，你必须告诉编译器你想要初始化的数组是什么类型：

```
MyMethod(new string[] {"Holly", "Jon", "Tom", "Robin", "William"});
```

C# 3允许介于这两者之间的一种写法：

```
MyMethod(new[] {"Holly", "Jon", "Tom", "Robin", "William"});
```

显然，编译器必须自己判断要使用什么类型的数组。它首先构造一个集合，其中包括大括号内所有表达式的编译时类型。在这个类型的集合中，如果其他所有类型都能隐式转换为其中一种类型，该类型即为数组的类型。否则（或者所有值都是无类型的表达式，比如不变的null值或者匿名方法，而且不存在强制类型转换），代码就无法编译。

注意，只有表达式的类型才会成为一个候选的数组类型。这意味着你偶尔需要将一个值显式转型为一个不太具体的类型。例如，以下代码无法编译：

```
new[] { new MemoryStream(), new StringWriter() }
```

不存在从MemoryStream向StringWriter的转换，反之亦然。两者都能隐式转换为object和IDisposable，但编译器只能在表达式本身产生的原始集合中过滤。在这种情况下，如果修改其中的一个表达式，把它的类型变成object或IDisposable，代码就可以编译了：

```
new[] { (IDisposable) new MemoryStream(), new StringWriter() }
```

最终，表达式的类型为IDisposable[]。当然，代码都写成这样了，还不如考虑一下显式声明数组的类型，就像在C# 1和C# 2中所做的那样，从而更清楚地表达你的意图。

与前面的特性相比，隐式类型的数组显得有点儿虎头蛇尾。反正我自己很难对它产生多大的兴趣——虽然在将数组作为参数传递时，它确实能稍微简化一下代码。我们知道，语言的设计者在决定一个特性是否应该成为语言的一部分时，必须考虑“有用性”和“复杂性”的平衡，这个

8

特性我感觉并没有平衡好。

但是，设计者推出这个特性也并不是“心血来潮”——在一个非常重要的场合中，隐式类型这个特性还是非常关键的。那就是在不知道（也无法知道）数组元素的类型的时候。怎么会出现这种稀奇古怪的情况？继续读下去……

8.5　匿名类型

如果分开来看，隐式类型、对象和集合初始化程序以及隐式数组类型均有或大或小的用处。然而，它们也可以组合起来使用，从而服务于一个更高的目标，也就是本章最后一个要讲述的特性——*匿名类型*。而匿名类型又服务于一个还要高的目标——LINQ。

8.5.1　第一次邂逅匿名类型

在通过一个例子获得对匿名类型的一些观感之后，再来解释它会容易许多。抱歉的是，由于不能使用扩展方法和Lambda表达式，所以本节的例子看起来有一点儿“牵强”，但这里存在着一个先有鸡还是先有蛋的问题。确实，匿名类型在与更高级的特性结合时最有用。但是，首先必须掌握一些基础知识，才能理解基于它建立起来的东西。坚持下去，我保证这里学到的东西从长远来说会是很有意义的。

假设现在没有`Person`类，而且我们唯一关心的属性就是名字和年龄。代码清单8-4展示了在不声明一个类型的前提下，如何构造具有这些属性的对象。

代码清单8-4　创建具有Name和Age属性的匿名类型的对象

```
var tom= new { Name = "Tom", Age = 9 };
var holly = new { Name = "Holly", Age = 36 };
var jon = new { Name = "Jon", Age = 36 } ;

Console.WriteLine("{0} is {1} years old", jon.Name, jon.Age);
```

从代码清单8-4可以看出，对匿名类型进行初始化的语法与8.3.2节展示的对象初始化程序的语法非常相似，只是在`new`和`{`之间没有了类型名称。之所以要用隐式类型的局部变量，是因为那是唯一能用的（当然，除了`object`外）——本来就没有一个明确的类型名称来声明变量。如最后一行所示，这个匿名的类型有`Name`和`Age`两个属性，两者都可读，其值已在用于创建实例的*匿名对象初始化程序*中指定。所以这一行输出的是`Jon is 36 years old`。属性具有与初始化程序中的表达式一样的类型——`Name`是`string`，而`Age`是`int`。和普通的对象初始化程序一样，在匿名对象的初始化程序中，表达式可以调用所有方法或构造函数，可以获取属性值，可以执行计算——可以做你需要做的任何事情。

现在，你对隐式类型的数组的重要性可能有所体会了。假定现在要创建一个数组来包含所有家庭成员，然后遍历它来计算所有人的总年龄[①]。代码清单8-5能完成这项工作，与此同时，它还

① 如果你知道LINQ，会感觉这是一种不恰当的计算年龄总数的方式。没错，调用`family.Sum(p => p.Age)`要更加简单明了——不过，我们还是一步一步来吧。

演示了匿名类型的其他一些有趣的特性。

代码清单8-5 用匿名类型填充数组，并计算总年龄

```
var family = new[]                                  ❶ 使用隐式类型的数组初始化程序
{
    new { Name = "Holly", Age = 36 },
    new { Name = "Jon", Age = 36 },
    new { Name = "Tom", Age = 9 },
    new { Name = "Robin", Age = 6 },                ❷ 同一个匿名类型连用5次
    new { Name = "William", Age = 6 }
};

int totalAge = 0;                                   ❸ 对每个人使用隐式类型
foreach (var person in family)
{
    totalAge += person.Age;                         ❹ 求年龄之和
}
Console.WriteLine("Total age: {0}", totalAge);
```

综合代码清单8-5和我们在8.4节学到的有关隐式类型的数组的知识，可以推断出非常重要的一点：`family`中的所有人都有相同的类型。如果❷中每次使用匿名对象初始化程序生成的是不同的类型，编译器就无法推断❶中声明数组的适当类型。在任何一个给定的程序集中，如果两个匿名对象初始化程序包含相同数量的属性，且这些属性具有相同的名称和类型，而且以相同的顺序出现，就认为它们是同一个类型。换言之，如果在某个初始化程序中交换了`Name`和`Age`的顺序，就会出现两个不同的类型。类似地，如果在某行中引入了一个额外的属性，或者某人的年龄使用的是`long`值而不是`int`值，就会引入一个新的匿名类型。这时，对数组所做的类型推断将会失败。

说明 **实现细节：有多少个类型？** 如果你决定查看微软编译器生成的匿名类型的IL（或者反编译的C#），注意以下情况：假定两个匿名对象初始化程序按照相同的顺序使用了相同的属性名称，但属性的类型不同，那么最终虽会产生两个不同的类型，它们实际上是从同一个泛型类型生成的。该泛型类型是参数化的，同时也是封闭的。由于不同的初始化程序会提供不同的类型实参，因此构造出来的类型是不同的[①]。

注意，可以用一个`foreach`语句来遍历数组，就像遍历其他集合一样。所涉及的类型是推断出来的❸，`person`变量的类型就是我们在数组中使用的匿名类型。同样，我们可以使用同一个变量表示不同的实例，因为它们全都具有相同的类型。

代码清单8-5还证明了`Age`属性真的是`int`这个强类型——如若不然，求年龄之和❹是通不过编译的。编译器了解匿名类型的一切，Visual Studio甚至能够通过工具提示来分享关于它的信息，以防你不确定。图8-4演示了将鼠标对准代码清单8-5中的表达式`person.Age`的`person`部分时出现的提示。

① “参数化的泛型类型”是指泛型类型的成员允许作为参数来传递。在“封闭的泛型类型”中，涉及的所有类型都是已知的，不涉及来自外部的一个类型参数。这些主题均在第3章进行了详细讨论。——译者注

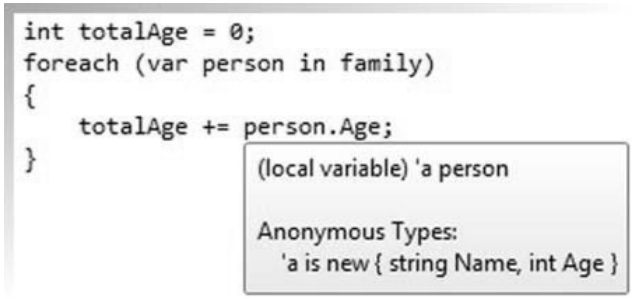


图8-4　鼠标对准（隐式）声明为匿名类型的一个变量时，会显示该匿名类型的细节

既然看到了匿名类型的实际应用，接着让我们回过头去研究一下编译器到底为我们做了哪些事情。

8.5.2　匿名类型的成员

匿名类型由编译器创建并包含到最终的程序集中。编译器还以同样的方式处理匿名方法和迭代器块所涉及的额外类型。CLR把它们看做普通的类型，它们也真的是普通的类型——如果以后决定将匿名类型全部改成普通的、手动编码的类型，并使它们具有本节描述的行为，那么不会发生任何改变。

匿名类型包含以下成员：

- 一个获取所有初始值的构造函数。参数的顺序和它们在匿名对象初始化程序中的顺序一样，名称和类型也一样；
- 公有的只读属性；
- 属性的私有只读字段；
- 重写的Equals、GetHashCode和ToString。

就这么多了——没有实现接口，没有克隆或序列化能力——就是一个构造函数、一些属性以及一些源自object的普通方法。

构造函数和属性所做的事情是显而易见的。同一个匿名类型的两个实例在判断相等性时，采用的是自然的方式：用属性类型的Equals方法来依次比较每个属性值。散列码的生成也是相似的：依次为每个属性值调用GetHashCode，并组合结果。至于具体是用什么方法来组合各个散列码以生成一个“复合”散列码，文档中没有指明，你的代码不应依赖于它——唯一能确定的是，两个相等的实例肯定会返回相同的散列码，两个不相等的实例通常返回不同的散列码。当然，要想得到这样的结果，对于属性所涉及的各个不同的类型，它们的Equals和GetHashCode的实现必须符合标准。

由于所有属性都是只读的，所以只要这些属性是不易变的，那么匿名类型就是不易变的。这就为你提供了“不易变”这一特性所具有全部常规性的优势——能放心向方法传递值，不用害怕这些值被改变；能在线程之间共享数据，等等。

> 说明　**VB中匿名类型的属性默认为易变的**　Visual Basic 9和Visual Basic 10中同样可以使用匿名类型，但默认情况下它们的属性是易变的，如果要声明不易变的属性，需要使用Key修饰符。只有作为键声明的属性才会用于散列和相等性比较。在将代码从一种语言转换到另一种语言时，很容易忽视这一点。

现在，我们已经差不多完成了对匿名类型的讨论。然而，还有一点需要注意，C# 3为LINQ中一种很常见的情形提供了一个简化的语法。

8.5.3　投影初始化程序

到目前为止，我们看到过的匿名对象初始化程序都是由名字/值对构成的一个列表，比如Name = "Jon", Age=36。前面一直用的都是常量，因为这样让例子显得比较小。在真正的代码中，你经常都需要从现有的对象复制属性。你有时希望以某种方式处理值，但通常一次直接的复制就足够了。

同样地，在不涉及LINQ的前提下，这里很难举出一个有说服力的例子。但是，让我们回到前面的Person类，并假定现在有一个很好的理由需要将Person实例的一个集合转换成一个相似的集合。在新集合中，每个元素只有一个name，另外还有一个标志值指出这个人是不是成年人。在给定了一个恰当的person变量后，接着可以使用如下代码：

```
new { Name = person.Name, IsAdult = (person.Age >= 18) }
```

这肯定能行，而且对单一个属性来说，设置名字的语法（加粗的部分）并不太笨拙。但是，假如要复制的是几个属性，代码看起来就有点儿“烦人”了。

C# 3支持一种简化的语法：如果不指定属性名称，而是只指定用于求值的表达式，它就会使用表达式的最后一个部分作为名称——前提是它只能是一个简单字段或属性。这就是所谓的*投影初始化程序*（projection initializer）。换言之，上述代码可重写为：

```
new { person.Name, IsAdult = (person.Age >= 18) }
```

很常见的一种情况是，一个匿名对象初始化程序中的每一个部分都是投影过来的。例如，一些属性从一个对象中获取，另一些属性从另一个对象中获取。这通常是一次连接操作的一部分。不管怎样，我们有点儿超前了。

代码清单8-6展示了上述代码的实际应用，其中使用了List<T>.ConvertAll方法和一个匿名方法。

代码清单8-6　从Person对象转换成一个名字和一个成年标志

```
List<Person> family = new List<Person>
{
   new Person { Name = "Holly", Age = 36 },
   new Person { Name = "Jon", Age = 36 },
   new Person { Name = "Tom", Age = 9 },
   new Person { Name = "Robin", Age = 6 },
   new Person { Name = "William", Age = 6 }
```

```
};
var converted = family.ConvertAll(delegate(Person person)
   { return new { person.Name, IsAdult = (person.Age >= 18) }; }
);
foreach (var person in converted)
{
   Console.WriteLine("{0} is an adult? {1}",
                     person.Name, person.IsAdult);
}
```

除了为Name属性使用了一个投影初始化程序，代码清单8-6还展现了委托类型推断和匿名方法的价值。如果没有它们，就没有办法保持converted的强类型，因为我们没有办法指定Converter的TOutput类型参数。如代码所示，我们可以遍历新列表并访问Name和IsAdult属性，这和访问其他任何类型无异。

暂时不要花太多的时间思考投影初始化程序的问题，只需记住它们确实存在，这样以后看到的时候才不会感到迷惑。事实上，这个忠告适用于整个匿名类型一节——下面让我们在不涉及细节的前提下，看一看它们到底因何而存在。

8.5.4　重点何在

此时此刻，希望你没有上当受骗的感觉，但如果有，我表示同情。对于我们还没有真正遇到的一个问题，匿名类型是一个相当复杂的解决方案，但我敢断定你以前确定见识过这个问题的一部分。

如果你从事过任何真正的、涉及数据库的编程工作，就知道假如指定一个查询条件，那么对于符合这个条件的所有行来说，你并非总想取回这些行中的所有数据。如果一次性获取超过自己需要的数据，这通常不是一个大问题，但假如你只需要表中总共50列数据中的2列，那么你肯定不愿意选择全部50列数据，是吧？

非数据库的代码存在同样的问题。假定用一个类来读取一个日志文件，并生成一系列带有多个字段的日志行。假如我们关心的只是日志中的几个字段，那么保留所有信息可能需要太多的内存。使用LINQ可轻松过滤那些信息。

但是，过滤之后的结果是什么呢？我们怎样保留一些数据，同时丢弃剩余数据呢？怎样轻松保留那些不能直接以原始形式来表示的派生数据呢？怎样合并那些起初没有联系或者那些仅在特定情况下才会产生联系的数据段呢？实际上，我们需要的是一个新的数据类型——但在每种不同的情况下都手动创建这样的一个类型显得太繁琐了，尤其是在已经有LINQ这样的辅助工具来简化剩余过程的前提下。图8-5展示了使匿名类型成为一个强大的C#语言特性的3个要素。

如果你要创建的一个类型只在一个方法中使用，而且其中只包含了字段和普通属性，就考虑一下能否使用匿名类型。我猜想，在你学习使用匿名类型的大多数时候，其实也可以考虑使用LINQ——这一点也要请你留意。

但是，假如需要使用同一个属性序列，在几个地方达到相同的目的，就应该考虑创建一个普通类型，即使其中只包含了普通的属性。匿名类型会通过隐式类型自然地“感染”使用它们的代

码——这通常不会带来问题，但在某些时候也可能令人生厌。特别是你无法简单地创建一个方法，使其以强类型的方式返回具有这个类型的实例。和前面讨论的其他特性一样，只有在真正会使代码变得更简单的时候才使用匿名类型——不要为了“尝鲜”和“耍酷”去用它。

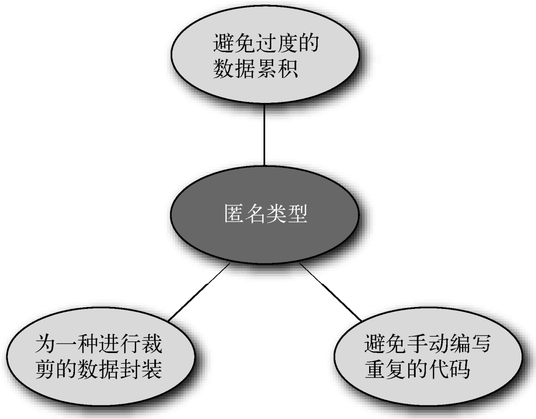


图8-5　匿名类型允许你只保留特定情况下需要的数据，针对这种情况进行裁剪，不必每次都单调重复地写一个新的类型

8.6　小结

看上去多像是特性的大杂烩！有4个特性是非常相似的，至少从语法上来说是这样的：对象初始化程序、集合初始化程序、隐式类型的数组以及匿名类型。其他两个特性——自动属性和隐式类型的局部变量则多少有点不同。同样，大多数特性在C# 2中单独使用时也很有用。但是，隐式类型的数组和匿名类型只有在与其他C# 3特性配合的时候才会体现它们的价值。

那么，这些特性有什么共同的地方呢？它们都能将开发人员从繁琐的编码中解脱出来！你肯定和我一样，不喜欢反复写那些普通得不能再普通的属性，也不喜欢用局部变量来一次设置一个属性——尤其是在构造一个由相似的对象构成的集合的时候。C# 3的新特性不仅使我们更容易写代码，还使代码变得更易读（至少是在运用得当的时候）。

下一章将探讨一个重要的新特性。与此同时，还会讨论它直接支持的一个框架特性。如果你认为匿名方法使创建委托变得容易了，那么等看到Lambda表达式的时候再来重新思考这个问题吧！

第 9 章

Lambda表达式和表达式树

本章内容

- Lambda表达式语法
- Lambda表达式转换成委托
- 表达式树框架类
- Lambda表达式转换成表达式树
- 表达式树的重要性何在
- 类型推断和重载决策发生的改变

通过第5章的学习，我们知道在方法组的隐式转换、匿名方法以及返回类型和参数逆变性的帮助下，C# 2极大地简化了委托的使用。如果仅仅是为了简化事件的订阅以及增强可读性，这些技术确实已经足够了。但是，C# 2中的委托仍然过于臃肿：一页充满匿名方法的代码读起来真让人难受，你也肯定不愿意经常在一个语句中放入多个匿名方法。

LINQ的基本功能就是创建操作管道，以及这些操作需要的任何状态。这些操作表示了各种关于数据的逻辑：如何过滤、如何排序以及如何将不同的数据源连接到一起，等等。当LINQ查询在“进程内”执行时，那些操作通常用委托来表示。

使用LINQ to Objects①来处理数据时，经常都会出现一个语句包含几个委托的情况。C# 3中的Lambda表达式在不牺牲可读性的前提下使这一切成为可能。（尽管我提到了可读性，但在本章中，Lambda表达式和Lambda是一个意思。）

> **说明** **我对此一窍不通** Lambda表达式”这个词来源于“Lambda演算”，后者通常也写作“λ演算”，其中λ是希腊字母，读作“Lambda”。λ演算涉及函数定义和函数应用的数学和计算机科学领域。它存在已经有很长一段时间，是包括ML在内的函数式编程语言（functional languages）的基础。幸好，你不需要知道λ演算就可以在C# 3中使用Lambda表达式。

① LINQ to Objects处理的是同一个进程中的数据序列。相比之下，像LINQ to SQL这样的provider将工作交给“进程外”的系统（比如数据库）去处理。

执行委托只是LINQ的众多能力之一。为了富有效率地使用数据库和其他查询引擎，我们需要以一种不同的方式来表示管道中的各个操作。即把代码当作可在编程中（programmatically）进行检查的数据。这样，操作的逻辑可转换成一种不同的形式，比如Web服务调用、SQL或LDAP查询等——什么合适就转换成什么。

虽然也可以在一个特殊的API中构造查询的表现形式，但这样的代码通常读起来比较费力，而且编译器也不好提供支持。这时Lambda表达式再一次大显神通：不仅可以用它们创建委托实例，而且C#编译器也能将它们转换成*表达式树*——用于表示Lambda表达式逻辑的一种数据结构，这样其他代码才能检查它。简言之，Lambda表达式用符合语言习惯的方式来表示LINQ数据管线中的操作。在后文中，我们将循序渐进。在拥抱整个LINQ之前，先分别讨论。

使用Lambda表达式的两种方式都会在本章讨论，但对表达式树的讨论还相当基础——我们暂时还不打算实际地创建任何SQL。然而，掌握了基本理论之后，等到第12章接触到真正令人印象深刻的东西时，你用起Lambda表达式和表达式树时就会相当顺手了。

本章最后一部分将讨论类型推断在C# 3中发生的变化，这些变化主要是由于某些Lambda表达式使用了隐式参数类型。这有点儿像学习怎样系鞋带：过程很无趣，但不学会的话，跑起来就有可能被绊倒。

我们先来看一下Lambda表达式是什么样子的。我们将从一个匿名方法开始，然后逐渐把它转换成更短的形式。

9.1　作为委托的 Lambda 表达式

从许多方面，Lambda表达式都可以看做是C# 2的匿名方法的一种演变。匿名方法能做的几乎一切事情都可以用Lambda表达式来完成[①]。另外，几乎在所有情况下，Lambda表达式都更易读、更紧凑。特别是，捕获的变量在Lambda表达式中的行为和在匿名方法中是一样的。从最显而易见的方面看，两者并无多大区别——只是Lambda表达式支持许多简化语法，使它们在常规条件下显得更简练。与匿名方法相似，Lambda表达式有特殊的转换规则：表达式的类型本身并非委托类型，但它可以通过多种方式隐式或显式地转换成一个委托实例。*匿名函数*这个术语同时涵盖了匿名方法和Lambda表达式——在很多情况下，两者可以使用相同的转换规则。

让我们先从一个非常简单的例子开始，它最初被表示成一个匿名方法。我们将创建一个委托实例，它获取一个`string`参数，并返回`int`（即字符串的长度）。首先需要选择一个委托类型。幸好，.NET 3.5提供了一整套泛型委托类型可以帮我们。

9.1.1　准备工作：`Func<...>`委托类型简介

在.NET 3.5的`System`命名空间中，有5个泛型`Func`委托类型。`Func`并无特别之处——只是它

① 匿名方法可以简明地忽略参数，但Lambda表达式不具备这一特性。详细的内容参见5.4.3节，不过在实践中，这算不上什么。

提供了一些好用的预定义泛型类型，在很多情况下能帮助我们处理问题。每个委托签名都获取0~4个参数，其类型是用类型参数[1]来指定的。最后一个类型参数用作每种情况下的返回类型。

以下是.NET 3.5中所有Func委托类型的签名：

```
TResult Func<TResult>()
TResult Func<T,TResult>(T arg)
TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

例如，Func<string,double,int>等价于以下形式的委托类型：

```
public delegate int SomeDelegate(string arg1, double arg2)
```

当你想返回void时，可使用Action<...>系列的委托，其功能相同。Action的单参数的版本在.NET 2.0中就有了，但其他都是.NET 3.5新增的。如果4个参数还嫌不够，.NET 4将Action<...>和Func<...>家族扩展为拥有16个参数，因此Func<T1, ..., T16,TResult>拥有让人欲哭无泪的17个类型参数。这主要是为了支持第14章要介绍的动态语言运行时（DLR），你不需要直接对它们进行处理。

例如，我们的示例类型需要获取一个string参数，并返回一个int，所以我们将使用Func<string,int>。

9.1.2　第一次转换成Lambda表达式

知道了委托类型后，接着就可以使用一个匿名方法来创建委托实例。代码清单9-1对此进行了演示。创建了委托实例后，我们还执行了这个委托实例，所以我们能看到它是如何工作的。

代码清单9-1　用匿名方法来创建委托实例

```
Func<string,int> returnLength;
returnLength = delegate (string text) { return text.Length; };

Console.WriteLine(returnLength("Hello"));
```

代码清单9-1最后会打印“5”，这是预料之中的。注意在上述代码中，returnLength的声明和赋值是分开的，否则一行可能放不下——除此之外，这样还有利于对代码的理解。匿名方法表达式用粗体显示，我们要将它转换成Lambda表达式。

Lambda表达式最冗长的形式是：

```
(显式类型的参数列表) => {语句}
```

=>部分是C# 3新增的，它告诉编译器我们正使用一个Lambda表达式。Lambda表达式大多数时候都和一个返回非void的委托类型配合使用——如果不返回一个结果，语法就不像现在这样一目了然。这标志着C# 1和C# 3在用法习惯上的另一个区别。在C# 1中，委托一般用于事件，很少会返回什么东西。而在LINQ中，它们通常被视为数据管道的一部分，接受输入并返回结果来表示投影的值，或者判断某项是否符合当前的过滤条件，等等。

① 你也许还记得在第6章我们使用过其中的一个版本，它没有任何参数（但有一个类型参数）。

这个版本包含了显式参数，并将语句放到大括号中，它看起来和匿名方法非常相似。代码清单9-2等价于代码清单9-1，只是使用了Lambda表达式。

代码清单9-2　冗长的第一个Lambda表达式，和匿名方法相似

```
Func<string,int> returnLength;
returnLength = (string text) => { return text.Length; };

Console.WriteLine(returnLength("Hello"));
```

同样，加粗部分是用于创建委托实例的表达式。在阅读Lambda表达式时，可以将=>部分看成“goes to”——所以代码清单9-2中的例子可以读成“`text` goes to `text.Length`”。由于这个部分暂时是我们唯一感兴趣的，所以从现在开始，我会把它单列出来。加粗的部分可以替换成本节列出的任何一个Lambda表达式，结果是相同的。

匿名方法中控制返回语句的规则同样适用于Lambda表达式：不能从Lambda表达式返回`void`类型；如果有一个非`void`的返回类型，那么每个代码路径都必须返回一个兼容的值①。所有这一切都非常直观，理解起来一点儿都不困难。

到目前为止，使用Lambda表达式并没有节省多大空间，或使代码变得多容易阅读。接下来我们将使用快捷语法。

9.1.3　用单一表达式作为主体

我们目前是用一个完整的代码块来返回值，这样可以灵活处理多种情况——可以在代码块中放入多个语句，可以执行循环，可以从代码块中的不同位置返回，等等。这和匿名方法是一样的。然而，大多数时候，都可以用一个表达式来表示整个主体，该表达式的值是Lambda的结果②。在这些情况下，可以只指定那个表达式，不使用大括号，不使用`return`语句，也不添加分号。格式随即变成：

```
(显式类型的参数列表) => 表达式
```

在我们的例子中，这意味着Lambda表达式变成了：

```
(string text) => text.Length
```

现在已经开始变简单了，接着来考虑一下参数类型。编译器已经知道`Func<string,int>`的实例获取单个字符串参数，所以只需命名那个参数就可以了。

9.1.4　隐式类型的参数列表

编译器大多数时候都能猜出参数类型，不需要你显式声明它们。在这些情况下，可以将Lambda表达式写成：

```
(隐式类型的参数列表) => 表达式
```

① 当然，那些抛出异常的代码路径（code path）不需要返回一个值，可检测的无限循环也不需要。

② 对于没有返回类型的委托，如果只有一条语句，也可以使用这种语法，省略分号和大括号。

隐式类型的参数列表就是一个以逗号分隔的名称列表，没有类型。但隐式和显式类型的参数不能混合匹配——要么整个列表都是显式类型的，要么全部都是隐式类型的。除此之外，如果有out或ref参数，就只能使用显式类型。但在我们的例子中，不存在这样的问题，所以我们的Lambda表达式可以继续简化成：

```
(text) => text.Length
```

这已经相当简短了——可以继续简化的地方已经不多了。不过，圆括号看起来是有点儿多余。

9.1.5　单一参数的快捷语法

如果Lambda表达式只需一个参数，而且那个参数可以隐式指定类型，C# 3就允许省略圆括号。这种格式的Lambda表达式是：

```
参数名 => 表达式
```

因此，我们的Lambda表达式的最终形式是：

```
text => text.Length
```

你也许会感到奇怪，Lambda表达式为什么有如此多的特殊情况。例如，在C#语言的其他任何地方，都不会关心一个方法是有一个参数还是有多个参数。事实上，现在看起来非常“特殊”的情况其实是*极其*普遍的。如果一小段代码含有多个Lambda，那么拿掉参数列表的圆括号之后，对于可读性的增强会是相当显著的。

值得注意的是，如果愿意，可以用圆括号将整个Lambda表达式括起来，就像其他表达式那样。将Lambda赋给一个变量或属性的时候，这样做也许能增强可读性——否则等号易使人混淆，至少开始时是这样。它们在大多数时候都是十分易读的，无须使用任何额外的语法。代码清单9-3基于我们的原始代码对此进行了演示。

代码清单9-3　一个简洁的Lambda表达式

```
Func<string,int> returnLength;
returnLength = text => text.Length;

Console.WriteLine(returnLength("Hello"));
```

代码清单9-3刚开始可能会让你觉得读起来有点“别扭”，这就跟当初许多开发者感到匿名方法很奇怪一样，不过他们很快就习惯了。在常规应用中，你会声明变量并在同一表达式中为它赋值，以使其更清晰。但当你习惯了Lambda表达式之后，一定会感慨它们是多么的简洁。很难想象还可以使用更短、更清晰的方式来创建委托实例[①]。可以将变量名text改成更短的名称，比如x——这在完整的LINQ中通常很有用——但较长的名称会为读者提供更多的信息。

我已经耗费了好几页篇幅来阐述这种转变，图9-1更加清晰地展示了我们究竟省略了多少无关的语法。

① 这样不是说不可能。有的语言允许用一个“魔术变量”（magic variable）来表示“只有单个参数”这种常见的情况，从而将闭包表示成简单的代码块。

是否为Lambda表达式的主体使用较短的形式（指定一个表达式来取代一个完整的代码块），以及是使用显式还是隐式参数，这两个决定是完全独立的。我们只是凑巧按上述顺序缩短Lambda表达式，但完全可以先从使用隐式参数开始。如果你已经熟悉了Lambda表达式，就不会再考虑这些问题了，你会自然而然地使用最简短的写法。

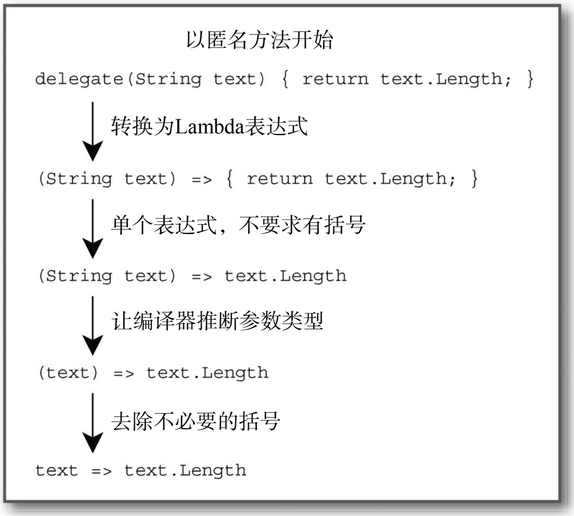


图9-1 Lambda语法简写

说明 **高阶（Higher-order）函数** Lambda表达式的主体本身可以包含另一个Lambda表达式，但做起来就像听起来一样，很容易使人混淆。另外，Lambda表达式的参数可以是另一个委托，这样做同样很乱。这两种情况都是高阶函数的例子。如果你喜欢这种茫然和困惑的感觉，可以去看看可下载的资源中的一些示例代码。虽然我对它们有一些微词，但这个做法在函数式编程中是十分常见的，而且可能非常有用。只不过，需要一定程度的坚持，才能习惯用它们来思考问题。

到目前为止，我们处理的只是单一的Lambda表达式，只是把它变换成不同的形式而已。在研究细节之前，让我们先来看几个具体的例子。

9.2 使用 `List<T>`和事件的简单例子

第10章讨论扩展方法的时候，我们会全部使用Lambda表达式。在那之前，最好的例子是`List<T>`和事件处理程序。首先从列表开始，为了使代码尽可能简洁，我们将使用自动实现的属性、隐式类型的局部变量以及集合初始化程序。然后，我们将调用以委托作为参数的方法——当然，是用Lambda表达式来创建委托。

9.2.1　列表的过滤、排序和操作

还记得`List<T>`的`FindAll`方法吗，它获取一个`Predicate<T>`，并返回一个新列表，包含原始列表中与谓词匹配的所有元素？`Sort`方法获取一个`Comparison<T>`，并相应地对列表进行排序。最后，`ForEach`方法获取一个`Action<T>`，对每个元素执行特定的行为。代码清单9-4使用Lambda表达式为上述每个方法提供委托实例。我们用多部电影的名称和发行年份作为示例数据。程序先打印原始列表，然后创建并打印只包含老电影的过滤列表。最后，我们按电影名称排序并打印原始列表。（顺便说一句，假如换用C# 1来做同样的事情，想想所有这些操作需要多使用多少代码吧！这是很有趣的事情。）

代码清单9-4　用Lambda表达式来处理一个电影列表

```
class Film
{
   public string Name { get; set; }
   public int Year { get; set; }
}
...
var films = new List<Film>
{
   new Film { Name = "Jaws", Year = 1975 },
   new Film { Name = "Singing in the Rain", Year = 1952 },
   new Film { Name = "Some like it Hot", Year = 1959 },
   new Film { Name = "The Wizard of Oz", Year = 1939 },
   new Film { Name = "It's a Wonderful Life", Year = 1946 },
   new Film { Name = "American Beauty", Year = 1999 },
   new Film { Name = "High Fidelity", Year = 2000 },
   new Film { Name = "The Usual Suspects", Year = 1995 }
};

Action<Film> print =                                    ❶ 创建可用的列表打印委托
   film => Console.WriteLine("Name={0}, Year={1}",
                             film.Name, film.Year);

films.ForEach(print);                                   ❷ 打印原始列表

films.FindAll(film => film.Year < 1960)                 ❸ 创建过滤过的列表
     .ForEach(print);

films.Sort((f1, f2) => f1.Name.CompareTo(f2.Name));     ❹ 对原始列表排序
films.ForEach(print);
```

代码清单9-4的前半部分是对数据进行设置。在本例中，使用匿名类型将不是一般的麻烦。所以简便起见，我创建了一个明确的类型。

在使用新建的列表之前，我们先创建好一个委托实例❶，用来打印列表中的项。这个委托实例会使用3次，这正是我创建了一个变量来保存它，而不是每次都单独使用一个Lambda表达式的原因。它的作用很简单，就是打印一个元素，但通过把它传给`List<T>.ForEach`，就可以轻松地将整个列表都打印到控制台上。很容易忽视但却十分重要的一点是，语句最后的分号是这个赋

值语句的一部分，而不是Lambda表达式的一部分。如果我们将同样的Lambda表达式作为方法调用的参数，就不会在Console.WriteLine(…)后面直接出现分号了。

我们打印的第一个列表❷是未作任何修改的原始列表。然后，我们找出列表中1960年以前制作的所有电影并打印出来❸。这是用另一个Lambda表达式来完成的，针对列表中的每一部电影，都会执行这个Lambda表达式，它只需判断一部电影是否应该包含到已过滤列表中。源代码将Lambda表达式作为一个方法实参来使用，但实际编译器会创建下面这样的方法：

```
private static bool SomeAutoGeneratedName(Film film)
{
   return film.Year < 1960;
}
```

所以调用FindAll的方法实际会变成：

```
films.FindAll(new Predicate<Film>(SomeAutoGeneratedName))
```

这里对Lambda表达式的支持就像是C# 2中对匿名方法的支持，智能的编译器在幕后指挥着一切操作。（事实上，微软的编译器在这种情况下要更智能一些——它知道以后假如代码再次被调用，可以重用委托实例，所以会把它缓存下来。）

排序列表❹也是用一个Lambda表达式来执行的，它比较任意两部电影的名称。必须承认的是，显式调用CompareTo使代码变得有点儿“难看”。下一章会介绍如何在OrderBy扩展方法的协助下，以一种更简洁的方式来表示排序。

下面来看一个例子，这一次使用Lambda表达式进行事件处理。

9.2.2 在事件处理程序中进行记录

9

在5.9节中，我们演示了如何利用匿名方法来方便地记录发生的事件。但是，之所以能使用这种简化的语法，完全是因为我们不介意丢失参数信息。假如我们既想记录事件的本质，又想记录与发送者和实参有关的信息①，又该怎么办呢？Lambda表达式允许我们以简洁的方式解决这个问题，如代码清单9-5所示。

代码清单9-5 使用Lambda表达式来记录事件

```
static void Log(string title, object sender, EventArgs e)
{
   Console.WriteLine("Event: {0}", title);
   Console.WriteLine(" Sender: {0}", sender);
   Console.WriteLine(" Arguments: {0}", e.GetType());
   foreach (PropertyDescriptor prop in
            TypeDescriptor.GetProperties(e))
   {
      string name = prop.DisplayName;
      object value = prop.GetValue(e);
      Console.WriteLine("    {0}={1}", name, value);
   }
}
```

① 也就是你熟悉的delegate (object sender, EventArgs e) {...};。——译者注

```
...
Button button = new Button { Text = "Click me" };
button.Click      += (src, e) => Log("Click", src, e);
button.KeyPress   += (src, e) => Log("KeyPress", src, e);
button.MouseClick += (src, e) => Log("MouseClick", src, e);

Form form = new Form { AutoSize = true, Controls = { button } };
Application.Run(form);
```

代码清单9-5使用Lambda表达式将事件名称和事件的参数传给记录事件细节的`Log`方法。除了由事件来源的`ToString`覆盖的版本返回的内容之外，我们没有记录事件来源的其他任何细节，这是由于与控件关联的信息实在是太多了。然而，我们对属性描述符（Property Descriptor）使用了反射技术，从而展示了传递的`EventArgs`实例的细节。

以下是单击按钮之后的一些示例输出：

```
Event: Click
 Sender: System.Windows.Forms.Button, Text: Click me
 Arguments: System.Windows.Forms.MouseEventArgs
   Button=Left
   Clicks=1
   X=53
   Y=17
   Delta=0
   Location={X=53,Y=17}
Event: MouseClick
 Sender: System.Windows.Forms.Button, Text: Click me
 Arguments: System.Windows.Forms.MouseEventArgs
   Button=Left
   Clicks=1
   X=53
   Y=17
   Delta=0
   Location={X=53,Y=17}
```

当然，不用Lambda表达式也能做到这一切——但用Lambda显得更简洁。

现在我们已经看到了Lambda表达式转换成委托实例的例子。接着，让我们研究一下表达式树，它将Lambda表达式表示成数据而不是代码。

9.3　表达式树

代码作为数据（code as data）是一个古老的概念，但它在流行的编程语言中使用得并不多。你可能会争辩说所有.NET程序都使用了这个概念，因为JIT将IL代码视为数据，并把它们转换成能在某个CPU上运行的本地代码。但这一切被深深地隐藏在幕后。另外，虽有一些库能在程序中操纵IL，但它们使用得并不广泛。

.NET 3.5的表达式树[1]提供了一种抽象的方式将一些代码表示成一个对象树。它类似于CodeDOM，但是在一个稍高的级别上操作。表达式树主要用于LINQ，本节稍后会解释表达式树

① 即Expression trees，文档中称为“表达式目录树”。——译者注

对于整个LINQ的重要性。

C# 3对于将Lambda表达式转换成表达式树提供了内建的支持。但在讨论这个问题之前，我们首先来看看在不使用任何编译器技术的情况下，它们是如何与.NET Framework融为一体的。

9.3.1 以编程方式构建表达式树

表达式树没有听起来那么神秘，虽然它们的一些用法使人感觉像在变魔术。顾名思义，它们是对象构成的树，树中每个节点本身都是一个表达式。不同的表达式类型代表能在代码中执行的不同操作：二元操作（例如加法），一元操作（例如获取一个数组的长度），方法调用，构造函数调用，等等。

`System.Linq.Expressions`命名空间包含了代表表达式的各个类，它们都继承自`Expression`，一个抽象的主要包含一些静态工厂方法的类，这些方法用于创建其他表达式类的实例。然而，`Expression`类也包括两个属性。

- `Type`属性代表表达式求值后的.NET类型，可把它视为一个返回类型。例如，如果一个表达式要获取一个字符串的`Length`属性，该表达式的类型就是`int`。
- `NodeType`属性返回所代表的表达式的种类。它是`ExpressionType`枚举的成员，包括`LessThan`、`Multiply`和`Invoke`等。仍然使用上面的例子，对于`myString.Length`这个属性访问来说，其节点类型是`MemberAccess`。

`Expression`有许多派生类，其中一些可能有多个不同的节点类型。例如，`BinaryExpression`就代表了具有两个操作数的任意操作：算术、逻辑、比较、数组索引，等等。这正是`NodeType`属性重要的地方，因为它能区分由相同的类表示的不同种类的表达式。

这里不打算讲述每个表达式类或节点类型——它们的数量实在是太多了，MSDN已经很好地解释了它们。相反，我会尝试让你对表达式树能做的事情有一个大致的感觉。

我们先来创建一个最简单的表达式树，让两个整数常量相加。代码清单9-6创建了一个表达式树来表示2+3。

代码清单9-6　一个非常简单的表达式树，2和3相加

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);

Console.WriteLine(add);
```

运行代码清单9-6会产生输出`"(2+3)"`，这意味着这些表达式树类覆盖了`ToString`来产生可读的输出。图9-2描绘了由上述代码生成的树。

值得注意的是，“叶”表达式在代码中是最先创建的：你自下而上构建了这些表达式。这是由“表达式不易变”这一事实决定的——创建好表达式后，它就永远不会改变。这样就可以随心所欲地缓存和重用表达式。

现在已经构建好了一个表达式树，下面来实际地执行它。

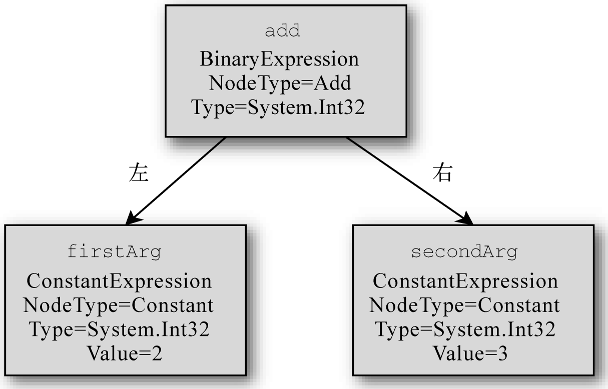


图9-2　代码清单9-6创建的表达式树的图形化表示

9.3.2　将表达式树编译成委托

LambdaExpression是从Expression派生的类型之一。泛型类Expression<TDelegate>又是从LambdaExpression派生的。这看起来有点儿令人迷惑——图9-3展示了其类型层次结构，使继承关系更加清晰。

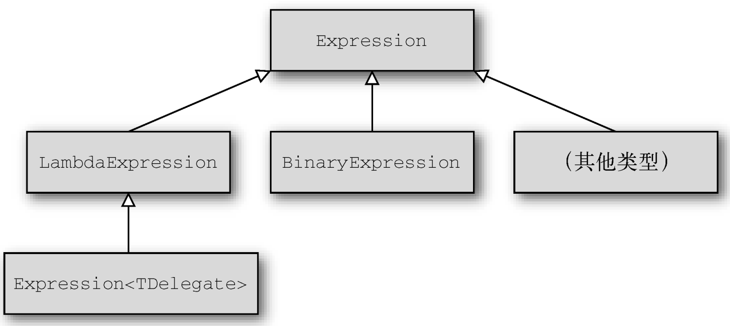


图9-3　从Expression<TDelegate>上溯至Expression的层次结构

Expression和Expression<TDelegate>类的区别在于，泛型类以静态类型的方式标识了它是什么种类的表达式，也就是说，它确定了返回类型和参数。很明显，这是用TDelegate类型参数来表示的，它必须是一个委托类型。例如，假设我们的简单加法表达式就是一个不获取任何参数，并返回整数的委托。与之匹配的签名就是Func<int>，所以可以使用一个Expression<Func<int>>，以静态类型的方式表示该表达式。我们用Expression.Lambda方法来完成这件事。该方法有许多重载版本——我们的例子使用的是泛型方法，它用一个类型参数来指定我们想要表示的委托的类型。其他备选方案请参见MSDN。

那么，这样做的意义何在呢？LambdaExpression有一个`Compile`方法能创建恰当类型的委托。`Expression<TDelegate>`也有一个同名的方法，但它静态类型化后返回`TDelegate`类型的委托。该委托现在可以采用普通方式执行，就好像它是用一个普通方法或者其他方式来创建的一样。代码清单9-7对此进行了演示，使用的是和以前一样的表达式。

代码清单9-7 编译并执行一个表达式树

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);

Func<int> compiled = Expression.Lambda<Func<int>>(add).Compile();
Console.WriteLine(compiled());
```

为了打印区区一个“5”，代码清单9-7或许是有史以来最为曲折的方式之一。但与此同时，它也给人留下了相当深刻的印象。我们在程序中创建了一些逻辑块，将其表示成普通对象，然后要求框架将所有的东西都编译成可以执行的“真实”的代码。你或许永远都不需要真正以这种方式使用表达式树，甚至永远都不需要在程序中构造它们，但它提供了相当有用的背景知识，可以帮助你理解LINQ是怎样工作的。

如本节开头所述，表达式树和CodeDOM差别不大——例如，Snippy（参见1.7.1节）就能编译和执行以普通文本形式输入的C#代码。但是，CodeDOM和表达式树之间存在着两个重要的区别。

首先，.NET 3.5的表达式树只能表示单一的表达式。它们不能表示完整的类、方法甚至语句。这在.NET 4中得到了一定的改善，表达式树可以支持动态类型——你可以创建块、为变量赋值，等等。但与CodeDOM相比，它仍然有着非常严格的限制。

其次，C#通过Lambda表达式直接在语言中支持表达式树。下面让我们具体来看一下。

9.3.3 将C# Lambda表达式转换成表达式树

我们知道，Lambda表达式能显式或隐式地转换成恰当的委托实例。然而，这并非唯一能进行的转换。还可以要求编译器通过你的Lambda表达式构建一个表达式树，在执行时创建`Expression<TDelegate>`的一个实例。例如，代码清单9-8展示了用一种精简得多的方式创建“返回5”的表达式，然后编译这个表达式，并调用编译得到的委托。

代码清单9-8 用Lambda表达式创建表达式树

```
Expression<Func<int>> return5 = () => 5;
Func<int> compiled = return5.Compile();
Console.WriteLine(compiled());
```

在代码清单9-8的第一行中，`() => 5`是Lambda表达式。在本例中，如果把它放到代码中一对额外的圆括号中，只会使它变得更难看而不是更好看。注意，我们不需要进行任何强制类型转换，因为编译器能判断这一切。可以用2+3来代替5，这样一来，编译器会进一步对这个加法运算进行优化。这里的一个要点在于，Lambda表达式已经转换成了一个表达式树。

注意　有一些限制　并非所有Lambda表达式都能转换成表达式树。不能将带有一个语句块（即使只有一个return语句）的Lambda转换成表达式树——只有对单个表达式进行求值的Lambda才可以。表达式中还不能包含赋值操作，因为在表达式树中表示不了这种操作。尽管.NET 4扩展了表达式树的功能，但只能转换单一表达式这一限制仍然有效。上述只是最常见的限制，还有另一些限制——完整的限制清单不值得在这里列出，因为其他限制所针对的问题极少出现。无论如何，假如试图进行的一个转换存在问题，你会在编译时发现。

下面来研究一个更复杂的例子，看看它们是如何工作的，并重点关注参数。这一次，我们要写一个获取两个字符串的谓词，并验证第1个字符串是否以第2个字符串开头。用Lambda表达式来表示是非常简单的，如代码清单9-9所示。

代码清单9-9　演示一个更复杂的表达式树

```
Expression<Func<string, string, bool>> expression =
   (x, y) => x.StartsWith(y);

var compiled = expression.Compile();

Console.WriteLine(compiled("First", "Second"));
Console.WriteLine(compiled("First", "Fir"));
```

表达式树本身则要复杂得多，尤其是等到我们把它转换成LambdaExpression的实例时。代码清单9-10展示了如何用代码来构造它。

代码清单9-10　用代码来构造一个方法调用表达式树

```
MethodInfo method = typeof(string).GetMethod
   ("StartsWith", new[] { typeof(string) });
var target = Expression.Parameter(typeof(string), "x");
var methodArg = Expression.Parameter(typeof(string), "y");
Expression[] methodArgs = new[] { methodArg };

Expression call = Expression.Call(target, method, methodArgs);

var lambdaParameters = new[] { target, methodArg };
var lambda = Expression.Lambda<Func<string, string, bool>>
   (call, lambdaParameters);

var compiled = lambda.Compile();

Console.WriteLine(compiled("First", "Second"));
Console.WriteLine(compiled("First", "Fir"));
```

❶ 构造方法调用的各个部件

❷ 从以上部件创建CallExpression

❸ 将Call转换成Lambda表达式

如你所见，代码清单9-10比使用C# Lambda表达式的那个版本涉及的东西多很多。然而，它确实更清晰地展示了树中涉及的东西以及参数是如何绑定的。为了构造最终的方法调用表达式，我们需要知道方法调用的几个部件❶，其中包括：方法的目标（也就是调用StartsWith的字符串）；方法本身（MethodInfo）；参数列表（本例只有一个参数）。在本例中，方法的目标和参

数恰好都是传递给表达式的参数，但它们完全可能是其他表达式类型，如常量、其他方法调用的结果、属性的求值结果，等等。

将方法调用构造成一个表达式之后❷，接着需要把它转换成Lambda表达式❸，并绑定参数。我们重用了作为方法调用（部件）信息而创建的参数表达式的值（`ParameterExpression`）：创建Lambda表达式时指定的参数顺序就是最终调用委托时使用的参数顺序。

图9-4展示了最终的表达式。这里有必要挑剔地说一下，虽然仍然称它为一个表达式树，但由于我们重用了参数表达式（只得如此——新建一个同名的参数表达式，并试图那样绑定参数，会在执行时出现一个异常），这意味着它不再是一个纯粹树了。

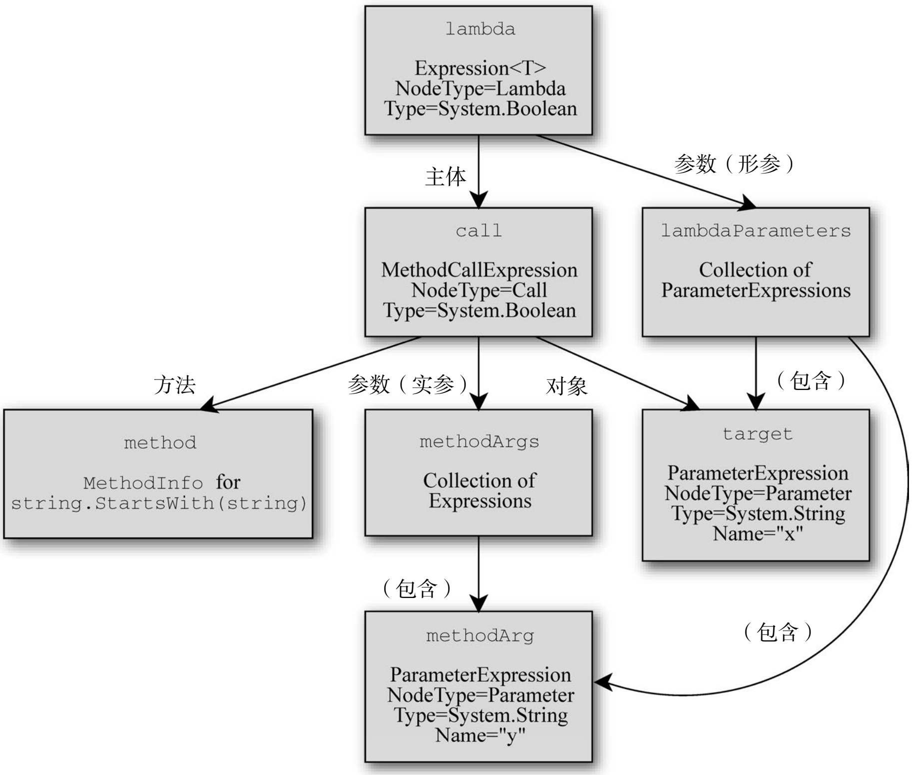


图9-4 用图形表示调用一个方法并使用来自一个Lambda表达式参数的表达式树

粗略地扫视一下图9-4和代码清单9-10的复杂程度，你就会发现，我们仅仅为了创建一个简单的方法调用，做了过于复杂的工作。可想而知，假如是一个相当复杂的表达式，与它对应的表达式树会复杂到什么程度。在这个时候，你会衷心感谢C# 3提供了从Lambda表达式创建表达式树的能力！

出于同样的目的，Visual Studio 2010为表达式树提供了一个内嵌的查看器（visualizer）[①]。如

① 如果你使用的是Visual Studio 2008，可以下载构建类似查看器的示例代码（参见http://mng.bz/g6xd），但如果你有Visual Studio 2010的话，使用其自带的查看器更简单一些。

果你想学习如何在代码中构建表达式树，或者想弄清楚它的内部结构，这个查看器非常有用。你可以使用一些虚拟数据编写一个Lambda表达式，在调试器中查看，然后弄明白如何用真实代码中的信息来构建类似的树。这个查看器依赖于.NET 4，因此不会对.NET 3.5的项目起作用。图9-5展示了StartsWith示例的查看结果。

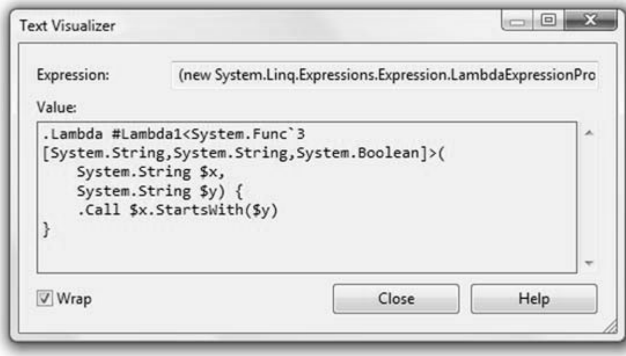


图9-5　在调试器中查看表达式树

查看器中的`.Lambda`和`.Call`部分分别对应于`Expression.Lambda`和`Expression.Call`调用。`$x`和`$y`对应于参数表达式。不管表达式树是使用显式代码构建的，还是使用Lambda表达式转换的，查看器中的结果都是相同的。

这里要注意的一个小问题是，虽然C# 3编译器在编译好的代码中会使用与代码清单9-10相似的代码来构造表达式树，但它有一个不为人所知的简便方法可以对代码进行优化：它并不需要使用普通的反射技术来获得`string.StartsWith`的`MethodInfo`。相反，它使用的是与`typeof`操作符等价的一个方法。该方法仅在IL中适用，在C#本身中是用不了的——相同的操作还用于从方法组创建委托实例。

现在，我们已经知道了表达式树和Lambda表达式是如何链接到一起的，接着简要地讨论一下它们为什么如此有用。

9.3.4　位于LINQ核心的表达式树

没有Lambda表达式，表达式树几乎没有任何价值。如果只想构建单个表达式而不是完整的语句、方法、类型等，它们可作为CodeDOM的一个替代方案来使用，但价值仍然有限。

从一定程度上说，反过来说也是成立的：没有表达式树，Lambda表达式肯定就没那么有用了。如果能用一种更简便的方式创建委托实例，并向更函数化的开发模式进行转移，那么何乐而不为？如下一章所述，在与扩展方法组合使用的时候，Lambda表达式的效率尤其高。不过，当表达式树介入后，事情就变得更有趣了。

那么，将Lambda表达式、表达式树和扩展方法合并到一起，会得到什么？答案是LINQ在C#语言这一层面的全部体现。第11章要讲述的额外语法可以说是锦上添花，但只需这3个角色故事

就足够精彩了。长期以来，我们要么能在编译时进行很好的检查，要么能指示另一个平台运行一些代码，这些指示一般表示成文本（如SQL查询）。但是，鱼和熊掌不可兼得。

Lambda表达式提供了编译时检查的能力，而表达式树可以将执行模型从你所需的逻辑中提取出来。将两者合并到一起之后，鱼和熊掌就能兼得了——当然是在一个合理的范畴之内。“进程外”LINQ提供器的中心思想在于，我们可以从一个熟悉的源语言（如C#）生成一个表达式树，将结果作为一个中间格式，再将其转换成目标平台上的本地语言，比如SQL。某些时候，你更多地会遇到一个本机API，而不是一种简单的本机语言。例如，这个API可能根据表达式所表示的内容来调用不同的Web服务。图9-6展示LINQ to Objects和LINQ to SQL的不同路径。

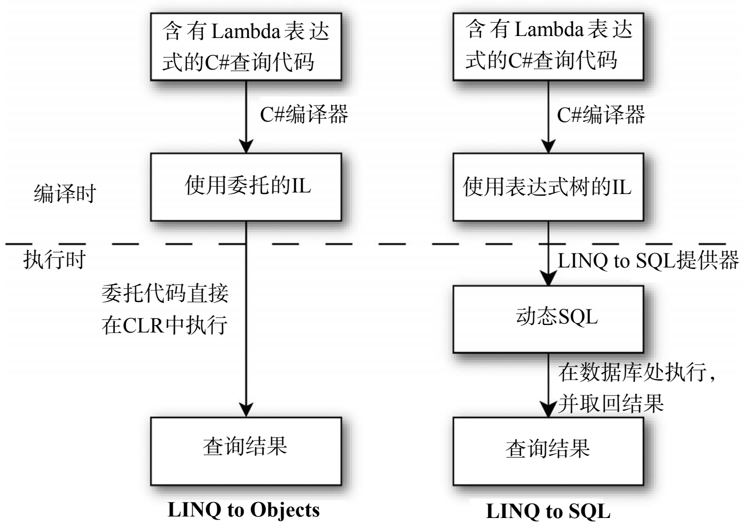


图9-6　无论LINQ to Objects还是LINQ to SQL都是始于C#代码，结束于查询结果。表达式树提供了远程执行代码的能力

一些情况下，转换会试图在目标平台上执行所有逻辑。而在其他情况下，则可能利用表达式树的编译功能在本地执行表达式的一部分，在别的地方执行另一部分。这个转换步骤的细节将在第12章讲述，但在第10章和第11章讨论扩展方法和LINQ语法的时候，你始终都应该记住这一终极目标。

说明　**编译器并不能做所有的检查**　表达式树被某些形式的转换器检查的时候，经常都会遇到被否决的情况。例如，虽然可以将`string.StartsWith`调用转换成类似的SQL表达式，但`string.IsInterned`调用在数据库环境中是没有意义的。表达式树确保了大量的编译时安全性，但编译器只能检查以确保Lambda表达式能转换成一个有效的表达式树，它不能保证表达式树最后的使用是否合适。

尽管表达式树常用于LINQ，但情况并不总是这样。

9.3.5　LINQ之外的表达式树

Bjarne Stroustrup曾经说过："我不会创建这样的工具，它所能做的，都是我可以想象得到的。"尽管表达式树主要是为LINQ而引入.NET的，但微软和社区都发现了它的一些其他用法。本节所列出的并不全面，但会帮助你理解表达式树可以做什么。

1. 优化动态语言运行时

我们在第14章讨论C#动态类型时，将看到更多关于动态语言运行时的内容。表达式树是其架构的核心部分。它们具有三个特点对DLR特别有吸引力：

- 它们是不易变的，因此可以安全地缓存；
- 它们是可组合的，因此可以在简单的块中构建出复杂的行为；
- 它们可以编译为委托，后者可以像平常那样进一步JIT编译为本地代码。

DLR需要对如何处理不同的表达式做出决定，这些表达式会因不同的规则而发生改变。表达式树允许将这些规则（和结果）转换为代码，这与你知道所有的规则和结果后，手工编写代码非常接近。这一概念异常强大，可以使动态代码以惊人的速度执行。

2. 可以放心地对成员的引用进行重构

在9.3.3节，我提到过编译器以与`typeof`操作符类似的方式生成到`MethodInfo`值的引用。然而C#却不具备相同的功能，也就是说要让一段通用的、基于反射的代码"使用我的类型中定义的`BirthDate`属性"，就只能使用字符串字面量，并确保在修改属性名称的时候，也要修改这个字面量。使用C# 3，你可以使用Lambda表达式构建一个代表属性引用的表达式树。方法随后会分析该表达式树，找出你要的属性。然后，它就可以根据这些信息，进行你想要的处理。当然你还可以将表达式树编译为委托，并直接使用。

如下面的代码所示：

```
serializationContext.AddProperty(x => x.BirthDate);
```

这样一来，序列化的上下文就知道你要序列化`BirthDate`属性，它会记录适当的元数据并获取它的值。序列化只是需要属性或方法引用的一种情况而已，它常常出现在反射驱动的代码中。现在，如果将`BirthDate`属性重构为`DateOfBirth`，Lambda表达式也会随之改变。当然，这还不是完全安全的，这里并没有对表达式是否真的对简单属性求值进行编译时检查，因此只能在`AddProperty`的代码中进行执行时检查。

总有一天C#会在语言层面获得这样的能力。这样的操作符已经起好名字了：infoof，可以读成"info-of"或"in-foof"，这取决于你的心情。它已经在C#团队可能的特性列表中存在了一段时间了，并且Eric Lippert还为此撰写了一篇文章（参见http://mng.bz/24y7）。但是它至今还没有付诸实践，也许会在C# 6中。

3. 更简单的反射

在深入类型推断之前，我最后要介绍的还是关于反射的用法。我在第3章提到过，算术操作符与泛型不能很好地结合使用，例如，你很难编写通用的代码将多个值相加。Marc Gravell使用

表达式树实现了一个泛型的Operator类和一个非泛型的辅助类，可以用来编写下面这样的代码：

```
T runningTotal = initialValue;
foreach (T item in values)
{
   runningTotal = Operator.Add(runningTotal, item);
}
```

即便values中的各值与runningToatal的类型不同，它仍然能正常工作。如将TimeSpan类型的序列与DateTime相加。在C# 2中也可以实现这个功能，但却相当繁琐，因为操作符会通过反射进行公开，特别是对于基元类型来说。而表达式树的实现则非常简洁，并且它们会编译成普通的IL，然后进行JIT编译，提供了很好的性能。

毫无疑问会有很多开发者正在使用完全不同的用法，这里列出的仅仅是一部分示例。我们与Lambda表达式和表达式树的直接对话到此告一段落。我们将在介绍LINQ时学习更多这方面的内容。但在继续深入之前，需要解释C#的一些改变。它们是关于类型推断以及编译器如何在多个重载方法之间进行选择。

9.4　类型推断和重载决策的改变

类型推断和重载决策所涉及的步骤在C# 3中发生了变化，以适应Lambda表达式，并使匿名方法变得更有用。这些虽然不算是C#的新特性，但在理解编译器所做的事情方面，这些变化是相当重要的。如果你感觉这样的细节过于繁琐且无关大局，不妨跳到本章最后的小结部分，但请记住本节的存在，以后假如遇到涉及该主题的编译错误，而且不能理解代码为何不能工作，请回到这里来寻找答案。（还有一个可能是，你本来并不认为自己的代码能通过编译，但它居然编译成功了，那么也可以回到这里来参考。）

本节并不打算面面俱到——那是C#语言规范的任务，位于C# 5规范的7.5.2节。我只是打算概述一下新的行为，提供常见情况下的例子。规范之所以发生了变化，是为了使Lambda表达式能够以一种简洁的方式工作，这正是我将这个主题放到本章讨论的原因。

让我们稍微深入地探讨一下假如C#团队坚守老的规则不变，将会遇到什么问题。

9.4.1　改变的起因：精简泛型方法调用

在几种情况下会进行类型推断。通过以前的讨论，我们知道隐式类型的数组以及将方法组转换为委托类型都需要类型推断，但将方法组作为其他方法的参数进行转换时，会显得极其混乱：要调用的方法有多个重载的方法，方法组内的方法也有多个重载方法，而且还可能涉及泛型泛型方法，一大堆可能的转换会使人晕头转向。

到目前为止，最常见的类型推断调用方法时不指定任何类型实参。在LINQ里面，这类事情是时刻都在发生的——查询表达式的工作方式严重依赖于此。这个过程被处理得如此顺畅，以至于很容易忽视编译器帮你做的大量工作，而这一切都是为了使你的代码更清晰、更简洁。

C# 2的类型推断规则颇为简单，虽然对方法组和匿名方法的处理得并非总如我们希望得那样好——类型推断过程不能从它们推断出任何信息，从而导致你想要的行为对开发者来说虽然是一

目了然的，但对编译器来说却并非如此。随着Lambda表达式的引入，C# 3中的情况变得更复杂——如果用一个Lambda表达式来调用一个泛型方法，同时传递一个隐式类型的参数列表，编译器就必须先推断出你想要的是什么类型，然后才能检查Lambda表达式的主体。

用实际的代码更容易说明问题。代码清单9-11列出了我们想解决的一类问题：用Lambda表达式调用一个泛型方法。

代码清单9-11　需要新的类型推断规则的例子

```
static void PrintConvertedValue<TInput,TOutput>
   (TInput input, Converter<TInput,TOutput> converter)
{
   Console.WriteLine(converter(input));
}
...
PrintConvertedValue("I'm a string", x => x.Length);
```

`PrintConvertedValue`方法直接获取一个输入的值和委托，将该值转换成不同类型的委托。它未就类型参数`TInput`和`TOutput`作出任何假设，因此完全是通用的。现在，让我们研究一下在本例中最后一行调用该方法时实参的类型到底是什么。第1个实参明显是字符串，但第2个呢？它是一个Lambda表达式，所以需要把它转换成一个`Converter <TInput,TOutput>`，而那意味着要知道`TInput`和`TOutput`的类型。

如果你记得的话（3.3.2节），就该知道C# 2的类型推断规则是单独针对每一个实参来进行的，从一个实参推断出的类型无法直接用于另一个实参。在当前这个例子中，这些规则会妨碍我们为第2个实参推断出`TInput`和`TOutput`的类型。所以，如果还是沿用C# 2的规则，代码清单9-11的代码就会编译失败。

本节的最终目标就是让你明白是什么使代码清单9-11在C# 3中成功通过编译（真的会成功，我保证），但让我们先从一些难度适中的内容入手。

9.4.2　推断匿名函数的返回类型

代码清单9-12展示了貌似能编译，但不符合C# 2类型推断规则的示例代码。

代码清单9-12　尝试推断匿名方法的返回类型

```
delegate T MyFunc<T>();                         <—— 声明了.NET2.0中没有的Func<T>

static void WriteResult<T>(MyFunc<T> function)  <—┐
{                                                 │ 声明带有委托参数的泛型方法
   Console.WriteLine(function());                 │
}                                                 ┘
...
WriteResult(delegate { return 5; });            <—— 要求对T进行类型推断
```

代码清单9-12在C# 2中编译会报错：

```
error CS0411: The type arguments for method
'Snippet.WriteResult<T>(Snippet.MyFunc<T>)' cannot be inferred from the
usage. Try specifying the type arguments explicitly.
```

可以采取两种方式修正这个错误：要么显式指定类型实参（就像编译器推荐的那样），要么将匿名方法强制转换为一个具体的委托类型：

```
WriteResult<int>(delegate { return 5; });
WriteResult((MyFunc<int>)delegate { return 5; });
```

这两种方式都可行，但看起来都有点儿令人生厌。我们希望编译器能像对非委托类型所做的那样，执行相同的类型推断，也就是根据返回的表达式的类型来推断T的类型。那正是C# 3为匿名方法和Lambda表达式所做的事情——但其中存在一个陷阱。虽然在许多情况下都只涉及一个return语句，但有时会有多个。

代码清单9-13是代码清单9-12稍加修改的一个版本，其中的匿名方法有时返回int，有时返回object。

代码清单9-13 根据一天当中的时间来选择返回int或object

```
delegate T MyFunc<T>();

static void WriteResult<T>(MyFunc<T> function)
{
   Console.WriteLine(function());
}
...
WriteResult(delegate
{
   if (DateTime.Now.Hour < 12)
   {
      return 10;             <—— 返回类型是int
   }
   else
   {
      return new object();        <—— 返回类型是object
   }
});
```

在这种情况下，编译器采用和处理隐式类型的数组时相同的逻辑来确定返回类型，详情可参见8.4节。它构造一个集合，其中包含了来自匿名函数主体中的return语句的所有类型[①]（本例是int和object），并检查是否集合中的所有类型都能隐式转换成其中的一个类型。int到object存在一个隐式转换（通过装箱），但object到int就不存在了。所以，object被推断为返回类型。如果没有找到符合条件的类型，或者找到了多个，就无法推断出返回类型，编译器会报错。

我们现在知道了怎样确定匿名函数的返回类型，但是，参数类型可以隐式定义的Lambda表达式又如何呢？

9.4.3 分两个阶段进行的类型推断

C# 3中的类型推断的细节比C# 2中的复杂得多。虽然你很少需要参考（C#语言）规范去

① 如果返回的表达式没有类型，比如是null或者是另一个Lambda表达式，就不会包含到这个集合中。其有效性将在以后进行检查——在决定了一个返回类型之后，但它们不参与那个决定。

了解确切的行为，但假如你真的需要，那么我建议你将所有类型参数、类型实参等都写到一张纸上，然后按照规范上所说的，一步一步地、细致地记录它要求的每一项操作。最后你将得到密密麻麻大量固定（fixed）和非固定（unfixed）类型变量，每个这样的变量都有一套不同的限制。固定类型变量是指编译器能确定其值的变量，否则就是非固定类型变量。而所谓限制，是指与一条与类型变量有关的信息。我猜你已经开始感到头痛了，如同我最开始接触这方面的知识一样。

在这里，我们打算以一种较为“笼统”的方式来思考类型推断——效果和你粗读一下C#语言规范差不多，但我们的方式会更容易理解。事实上，假如编译器不能完全按照你希望的方式执行类型推断，最后几乎肯定会造成一个编译错误，而不会生成一个行为不正确的程序。如果你的代码未能成功编译，请尝试向编译器提供更多的信息——就那么简单。然而，我下面仍然要大致地解释一下C# 3发生的改变。

第一个巨大的改变是所有方法实参在C# 3中是一个“团队”整体。在C# 2中，每个方法实参都被单独用于尝试确定一些类型参数。针对一个特定的类型参数，如果根据两个方法实参推断出不同的结果，编译器就会报错——即使推断结果是兼容的。但在C# 3中，实参可提供一些信息——被强制隐式转换为具体类型参数的最终固定变量的类型。用于推断固定值所采用的逻辑与推断返回类型和隐式类型的数组是一样的。

代码清单9-14展示了一个例子——没有使用任何Lambda表达式，就连匿名方法都没用。

代码清单9-14　综合来自多个实参的信息，灵活地进行类型推断

```
static void PrintType<T>(T first, T second)
{
   Console.WriteLine(typeof(T));
}
...
PrintType(1, new object());
```

在C# 2中，代码清单9-14的代码虽然在语法上是有效的，但并不能成功编译：类型推断会失败，因为从第一个实参判断出`T`肯定是`int`，第二个判断出`T`肯定是`object`，两个就冲突了。但在C# 3中，编译器会和在代码清单9-13中推断返回类型一样，判断出`T`应该是`object`。事实上，C# 3的类型推断过程现在已变得更加全面，推断返回类型时所采用的规则就是其中一个具有代表性的例子。

第二个改变在于，类型推断现在是分两个阶段进行的。第一个阶段处理的是“普通”的实参，其类型是一开始便知道的。这包括那些参数列表是显式类型的匿名函数。

稍后进行的第二个阶段是推断隐式类型的Lambda表达式和方法组的类型。其思想是，根据我们迄今为止拼凑起来的信息，判断是否足够推断出Lambda表达式（或方法组）的参数类型。如果能，编译器就可以检查Lambda表达式的主体并推断返回类型——这个返回类型通常能帮助我们确定当前正在推断的另一个类型参数。如果第二个阶段提供了更多的信息，就重复执行上述过程，直到我们用光了所有线索，或者最终推断出涉及的所有类型参数。

图9-7用流程图展示了这一过程，不过请记住，这只是该算法极度简化后的版本。

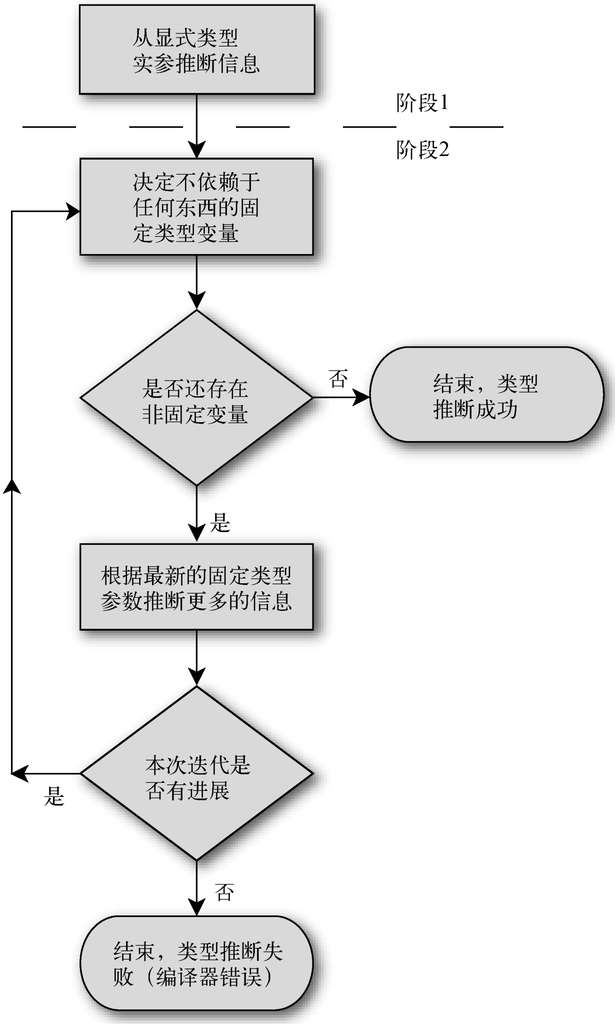


图9-7　类型推断流程的两个阶段

让我们用两个例子来展示这个过程。下面使用了本节最开始的代码清单9-11的代码：

```
static void PrintConvertedValue<TInput,TOutput>
   (TInput input, Converter<TInput,TOutput> converter)
{
   Console.WriteLine(converter(input));
}
...
PrintConvertedValue("I'm a string", x => x.Length);
```

代码清单9-11需要推断的类型参数是`TInput`和`TOutput`。具体步骤如下。

(1) 阶段1开始。

(2) 第1个参数是TInput类型，第1个实参是string类型。我们推断出肯定存在从string到TInput的隐式转换。

(3) 第2个参数是Converter<TInput,TOutput>类型，第2个实参是一个隐式类型的Lambda表达式。此时不执行任何推断，因为我们没有掌握足够的信息。

(4) 阶段2开始。

(5) TInput不依赖任何非固定的类型参数，所以它被确定为string。

(6) 第2个实参现在有一个固定的输入类型，但有一个非固定的输出类型。我们可把它视为(string x) => x.Length，并推断出其返回类型是int。因此，从int到TOutput必定会发生一个隐式转换。

(7) 重复"阶段2"。

(8) TOutput不依赖任何非固定的类型参数，所以它被确定为int。

(9) 现在没有非固定的类型参数了，推断成功。

复杂吗？确实有点复杂，但它很好地完成了工作——结果是我们希望的(TInput= string, TOutput=int)，一切都能正常编译，不会报错。

下一个例子更好地展示了重复阶段2的重要性。代码清单9-15执行了两个转换，第1个的输出成为第二个的输入。在推断出第一个转换的输出类型之前，我们不知道第二个的输入类型，所以也不能推断出它的输出类型。

代码清单9-15　多级类型推断

```
static void ConvertTwice<TInput,TMiddle,TOutput>
    (TInput input,
     Converter<TInput,TMiddle> firstConversion,
     Converter<TMiddle,TOutput> secondConversion)
{
    TMiddle middle = firstConversion(input);
    TOutput output = secondConversion(middle);
    Console.WriteLine(output);
}
...
ConvertTwice("Another string",
             text => text.Length,
             length => Math.Sqrt(length));
```

要注意的第一件事是方法签名看起来相当恐怖，但当你不再害怕，并仔细观察它时，发现它也没那么恐怖——当然示范用法使它看上去更直观。我们获取一个字符串，对它执行一次转换：这个转换和之前的转换是相同的，只是一次长度计算。然后，我们获取长度（int），并计算它的平方根（double）。

类型推断的"阶段1"告诉编译器肯定存在从string到TInput的一个转换。第一次执行"阶段2"时，TInput固定为string，我们推断肯定存在从int到TMiddle的一个转换。第二次执行"阶段2"时，TMiddle固定为int，我们推断肯定存在从double到TOutput的一个转换。第三次执行"阶段2"时，TOutput固定为doluble，类型推断成功。当类型推断结束后，编译器就可

以正确地理解Lambda表达式中的代码。

说明 检查Lambda表达式的主体 Lambda表达式的主体只有在输入参数的类型已知之后才能进行检查。如果x是一个数组或字符串，那么Lambda表达式`x => x.Length`就是有效的，但在其他许多情况下它是无效的。当参数类型是显式声明的时候，这并不是一个问题，但对于一个隐式（类型的）参数列表，编译器就必须等待，直到它执行了相应的类型推断之后，才能尝试去理解Lambda表达式的含义。

这些例子每次只展示了一个改变——在实际应用中，围绕不同的类型变量可能产生多个方面的信息，这些信息可能是在不同的重复阶段发现的[①]。为了避免你（和我）绞尽脑汁，我决定不再展示任何更复杂的例子了——你只需理解常规的机制就可以了，即使确切的细节可能仍然是模模糊糊的也不要紧。

虽然这种情况表面上非常罕见，似乎不值得为其设立如此复杂的规则，但它在C# 3中实际是非常普遍的，尤其是对LINQ而言。事实上，在C# 3中，你可以在不用思考的情况下大量地使用类型推断——它会成为你的一种习惯。然而，如果推断失败，你就会奇怪为什么。届时，你可以重新参考本节的内容以及语言规范。

还有一个改变需要讨论，但听到下面的话，你会很高兴，这个改变比类型推断简单：方法重载。

9.4.4 选择正确的被重载的方法

如果多个方法的名字相同但签名不同，就会发生重载。有时，具体该用哪个方法是显而易见的，因为只有它的参数数量是正确的，或者只有用它，所有实参才能转换成对应的参数类型。

但是，假如多个方法看起来都合适，就比较麻烦了。7.5.3节规范中的具体规则相当复杂（不好意思又来了）——但关键在于每个实参类型转换成参数类型的方式[②]。例如，假定有以下方法签名，似乎它们是在同一个类型中声明的：

```
void Write(int x)
void Write(double y)
```

`Write(1.5)`的含义显而易见，因为不存在从`double`到`int`的隐式转换，但`Write(1)`对应的调用就麻烦一些。由于存在从`int`到`double`的隐式转换，所以以上两个方法似乎都合适。在这种情况下，编译器会考虑从`int`到`int`的转换，以及从`int`到`double`的转换。从任何类型“转换成它本身”被认为好于“转换成一个不同的类型”。这个规则称为“更好的转换”规则。所以对

① 作者所说的“改变”是指本节所描述的C# 3与C# 2相比，在类型推断上的两个改变，一个改变是方法实参协同确定最后的类型实参，另一个改变是类型推断现在分两个阶段进行。但这些改变并不是孤立的，而是相互联系，共同发挥作用的。但是，作者前面的例子并没有反映出这一点，他的每个例子只是展示了其中的一个改变。
——译者注

② 假设所有的方法都声明在相同的类中。因为如果涉及继承，将会更加复杂。并且这部分在C# 3中并没有发生改变。

于这种特殊的调用，Write(int x)方法被认为好于Write(double y)。

如果方法有多个参数，编译器需要确保存在最适合的方法。如果一个方法所涉及的所有实参转换都至少与其他方法中相应的转换“一样好”，并且至少有一个转换严格优于其他方法，我们就认为这个方法要比其他方法好。

现给出一个简单的例子，假定现在有以下两个方法签名：

```
void Write(int x, double y)
void Write(double x, int y)
```

对Write(1, 1)的调用会产生歧义，编译器会强迫你至少为其中的一个参数添加强制类型转换，以明确你想调用的是哪个方法。每个重载都有一个更好的实参转换，因此都不是最好的。

同样的逻辑在C# 3中仍然适用，但额外添加了与匿名函数[①]有关的一个规则（匿名函数永远不会指定一个返回类型）。在这种情况下，推断的返回类型（如9.4.2节所述）在“更好的转换”规则中使用。

下面来看一个需要新规则的例子。代码清单9-16包含两个名为Execute的方法，另外还有一个使用了Lambda表达式的调用。

代码清单9-16　委托返回类型影响了重载选择

```
static void Execute(Func<int> action)
{
   Console.WriteLine("action returns an int: " + action());
}
static void Execute(Func<double> action)
{
   Console.WriteLine("action returns a double: " + action());
}
...
Execute(() => 1);
```

在代码清单9-16中，对Execute的调用可以换用一个匿名方法来写，也可以换用一个方法组——不管以什么方式，凡是涉及转换，所应用的规则都是一样的。那么，最后会调用哪个Execute方法呢？重载规则指出，在执行了对实参的转换之后，如果发现两个方法都合适，就对那些实参转换进行检查，看哪个转换“更好”。这里的转换并不是从一个普通的.NET类型到参数类型，而是从一个Lambda表达式到两个不同的委托类型。那么，哪个转换“更好”？

令人吃惊的是，同样的情况如果在C# 2中发生，那么会导致一个编译错误——因为没有针对这种情况的语言规则。但在C# 3中，最后会选中参数为Func<int>的方法。额外添加的规则可以表述如下：

> 如果一个匿名函数能转换成参数列表相同，但返回类型不同的两个委托类型，就根据从“推断的返回类型”到“委托的返回类型”的转换来判定哪个委托转换“更好”。

如果不拿一个例子来作为参考，这段话会绕得你头晕。让我们回头研究一下代码清单9-16：现

① 前面说过，匿名函数是匿名方法和Lambda表达式的统称。——译者注

在是从一个无参数、推断返回类型为int的Lambda表达式转换成Func<int>或Func<double>。两个委托类型的参数列表是相同的（空），所以上述规则是适用的。然后，我们只需判断哪个转换“更好”就可以了：int到int，还是int到double。这样就回到了我们熟悉的问题上——如前所述，int到int的转换更好。因此，代码清单9-16会在屏幕上显示：action returns an int: 1。

9.4.5 类型推断和重载决策

本节涵盖的内容比较多。我是愿意把它变得更简单一些的，但这本身就是一个很复杂的主题。另外，涉及的术语使内容更难以理解，尤其是参数类型和类型参数是全然不同的两样东西！如果你通读了本节的内容，而且真正理解了，那么我恭喜你！如果还有不明白的地方，也不必担心：希望下次重读本节的内容时，你能有更多的心得——尤其是真正遇到了对自己的代码起重要作用的问题时。现在总结一下本节的重点：

- 匿名函数（匿名方法和Lambda表达式）的返回类型是根据所有return语句的类型来推断的；
- Lambda表达式要想被编译器理解，所有参数的类型必须为已知；
- 类型推断不要求根据不同的（方法）实参推断出的类型参数的类型完全一致，只要推断出来的结果是兼容的就好；
- 类型推断现在分阶段进行，为一个匿名函数推断的返回类型可作为另一个匿名函数的参数类型使用；
- 涉及匿名函数时，为了找出“最好”的重载方法，要将推断的返回类型考虑在内。

虽然这是个短短的列表，但也可能令人望而生畏，因为里面涉及的技术术语太多。我再说一次，就算弄不明白也不要气馁。以我的经验来看，在多数时候，事情还是会朝着你希望的方向发展的。

9.5 小结

在C# 3中，Lambda表达式几乎完全取代了匿名方法。当然，为了保持向后兼容，匿名方法仍是支持的。但在符合C# 3语言习惯的、新写的代码中，几乎不会存在匿名方法。

然而，通过本章的描述，我们知道Lambda表达式并非仅仅是创建委托的一种更精简的语法。它们能转换成表达式树，然后可由其他代码处理，从而在不同的执行环境中执行等价的行为。如果没有这项功能，LINQ就仅限于进程内查询。

从某种程度上说，我们对类型推断和重载的讨论是迫不得已的：没有谁喜欢讨论那些必需的规则，但对所发生的事情至少有大致的了解，又是非常重要的。但在开始抱怨之前，请先想一想那些可怜的语言设计者，他们每天做的就是这类事情，确保所有规则都是一致的，而且在最恶劣的情况中也不会土崩瓦解。还有那些可怜的测试人员，他们必须在搞清楚所有规则之后，千方百计地瓦解设计者的实现！

对Lambda表达式本身的描述就是这么多了，但在本书剩余的部分，你还会看到更多的Lambda表达式。例如，下一章是完全围绕*扩展方法*展开的。从表面看，它们是完全独立于Lambda表达式的，但两个特性实际是经常在一起使用的。

第 10 章

扩展方法

10

本章内容

- 编写扩展方法
- 调用扩展方法
- 方法链
- .NET 3.5中的扩展方法
- 扩展方法的其他用途

我不是继承的粉丝。或者这样说，在我维护的代码中，或在我使用的类库中，有许多使用了继承的地方都是我不喜欢的。和其他许多东西一样，若使用得当，它会很强大，但其设计上的开销往往被人忽视，长此以往会给使用者带来痛苦。有时，我们用继承为一个类添加额外的行为和功能，即使并没有添加真正与对象有关的信息——此时没有什么需要特别说明。

继承有时是适宜的——如果新类型的对象应当携带有关额外行为的细节——但它多数时候都是不适宜的。事实上，许多时候根本不可能以这种方式使用继承，比如要处理的是值类型、密封类或者接口时。其他方案通常是写一堆静态方法，大多数静态方法都以目标类型的实例作为其中一个或者多个参数。虽然这很有效，而且不会产生因继承而带来的开销，但会使代码变得很难看。

C# 3引入了*扩展方法*的概念，它既有静态方法的优点，又使调用它们的代码的可读性得到了提高。使用扩展方法，可以像调用实例方法那样调用静态方法。不要惊慌——实际情况并不像听起来那样疯狂或者随便。

本章首先探讨如何使用和编写扩展方法，然后介绍.NET 3.5提供的一些扩展方法，并讨论如何将它们轻松链接到一起。这种链接（chaining）能力是最初在语言中引入扩展方法的重要原因之一，也是LINQ中相当重要的组成部分[1]。最后我们探讨扩展方法较之“普通”静态方法有哪些优缺点。

但是，首先还是让我们来看一看为什么在有的情况下，扩展方法与C# 1和C# 2中的静态方法相比是更理想的解决方案，尤其是在创建工具类的时候。

---

① 也许你已经听腻了有多少特性号称是“LINQ重要的组成部分”，这不怪你。这正是它伟大的地方之一。这些很小的部分单独拿出来似乎并不起眼，但组合起来却熠熠生辉。而每个特性也可以独立使用，但这其实是一个附加的好处。

10.1 未引入扩展方法之前的状态

这个时候你可能会产生似曾相识的感觉，因为工具类在第7章讲静态类的时候就出现过。如果你在开始用C# 3的时候，已经写了大量的C# 2代码，那么你应该看一看自己的静态类——其中许多方法可能都适合转换成扩展方法。我并不是说现有的所有静态类都适合，但你肯定能识别下面这些特征：

- 你想为一个类型添加一些成员；
- 你不需要为类型的实例添加任何更多的数据；
- 你不能改变类型本身，因为是别人的代码。

一个稍有变化的情况是你想处理接口而不是类。在这种情况下，你想添加一些有用的行为，这些行为只会调用到接口已经定义的方法①。例如`IList<T>`。如果能对`IList<T>`的任何（易变的）实现进行排序，岂不是很妙？但这会强迫接口的所有实现都必须实现自己的排序机制，这当然是很可怕的。不过从列表用户的角度来看，这的确是很妙的方法。

实际情况是，`IList<T>`提供了实现一个（实际是几个）完全通用的排序过程所需的全部构造块（building block）。但是，你不能将它放到接口中。实际上`IList<T>`应该设计成一个抽象类，并以这种方式包含排序功能。但是，由于C#和.NET只允许对实现进行单一继承，所以那样做会严重限制从这个抽象类派生的类型。使用扩展方法后，我们可对任何`IList<T>`的实现进行排序，而且感觉就像是列表本身提供的功能。

以后会看到，LINQ的许多功能都是围绕接口上的扩展方法建立起来的。但就目前来说，让我们暂时在后面的例子中使用一个不同的类型：`System.IO.Stream`。`Stream`类是.NET中的二进制通信的基础。`Stream`本身是抽象类，它有几个具体的派生类，比如`NetworkStream`、`FileStream`和`MemoryStream`。遗憾的是，有几个能带来方便的功能本应包含在`Stream`中，但实际上却被遗漏了。

在这些“遗漏的功能”中，我经常需要的是将整个流作为一个字节数组读入内存，以及将一个流的内容复制②到另一个流。这两个功能经常被人不恰当地实现，对流做出一些错误的假设，最常见的误解就是流中的数据如果没有先耗尽，那么`Stream.Read`会将给定的缓冲区填满。

说明　其实并没有完全“遗漏”　.NET 4添加了其中一个特性：`Stream`现在包含一个`CopyTo`方法。这也证明扩展方法有其脆弱的一面——我们将在10.2.3节讨论这一点。`ReadFully`依然被遗漏了，但毕竟使用这种方法需要十分谨慎：你只有确定流包含结尾，并且数据适合存入内存时，才能读取全部流。要知道，流的数据很可能是无穷无尽的。

一个理想的方案是将这些功能集中到一个地方，而不是在几个项目中都重复这些代码。这正是我要在自己的杂项工具库中写`StreamUtil`类的原因。代码清单10-1展示了一个精简的版本，

① 言外之意，不会涉及任何接口实现类的细节。——译者注

② 考虑到流的本质，这里的“复制”并不是指复制数据，而是从一个流读取数据，并把它写入另一个流。虽然从严格的意义上说，这里使用“复制”一词并不准确，但这一点点区别通常是无关紧要的。

10

它足够满足我们的要求，虽然在真实的代码中，还应包含大量的错误检查机制并包含其他功能。

代码清单10-1　为流提供附加功能的一个简单的工具类

```
using System.IO;

public static class StreamUtil
{
   const int BufferSize = 8192;

   public static void Copy(Stream input, Stream output)
   {
      byte[] buffer = new byte[BufferSize];
      int read;
      while ((read = input.Read(buffer, 0, buffer.Length)) > 0)
      {
         output.Write(buffer, 0, read);
      }
   }
   public static byte[] ReadFully(Stream input)
   {
      using (MemoryStream tempStream = new MemoryStream())
      {
         Copy(input, tempStream);
         return tempStream.ToArray();
      }
   }
}
```

实现细节并不太重要，你只需注意`ReadFully`方法调用了`Copy`方法——这对于稍后展示扩展方法的一个要点是非常有用的。这个类很容易使用——代码清单10-2展示了如何将Web响应写入磁盘。

代码清单10-2　用StreamUtil将Web响应流复制到一个文件

```
WebRequest request = WebRequest.Create("http://manning.com");
using (WebResponse response = request.GetResponse())
using (Stream responseStream = response.GetResponseStream())
using (FileStream output = File.Create("response.dat"))
{
   StreamUtil.Copy(responseStream, output);
}
```

代码清单10-2非常精简，`StreamUtil`类负责循环，每次都向响应流（response stream）索取更多的数据，直至接收到所有数据。虽然它很好地完成了一个工具类的工作，但“面向对象”的味道还是淡了一点。我们真正想要的是让响应流将它自身复制到`output`流，就像`MemoryStream`类的`WriteTo`方法那样。这个问题不大，只是代码看起来丑了一点。

继承在这个时候帮不了任何忙（我们希望这个行为对所有流都适用，而不仅是我们负责维护的流），而且不能更改`Stream`类本身——那么应该怎么办呢？如果是C# 2，我们没有任何办法——只能像上面那样使用静态方法，并忍受“丑陋”的代码。C# 3则允许我们更改静态类，把它的成员作为扩展方法显示出来，从而假装方法是`Stream`“与生俱来”的一部分。让我们看看需要进行哪些更改。

10.2 扩展方法的语法

扩展方法创建起来是非常容易的，使用起来也很简单，与最初学习写扩展方法时遇到的困难相比，使用它们的时机以及方式才是最值得我们考虑的具有重大意义的部分。先让我们转换前面的StreamUtil类来获得两个扩展方法。

10.2.1 声明扩展方法

并不是任何方法都能作为扩展方法使用——它必须具有以下特征：

- 它必须在一个非嵌套的、非泛型的静态类中（所以必须是一个静态方法）；
- 它至少要有一个参数；
- 第一个参数必须附加this关键字作为前缀；
- 第一个参数不能有其他任何修饰符（比如out或ref）；
- 第一个参数的类型不能是指针类型。

就这么多要求——方法可以是泛型的，可以有返回值，除第一个参数之外的其他参数可以是ref/out参数，可以用迭代块来实现，可以是分部类的一部分，可以使用可空类型——可以是任何方法，只要符合上述限制即可。

我们将第一个参数的类型称为方法的*扩展类型*（extended type），即指该方法扩展了该类型——在本例中我们扩展了Stream。这不是语言规范中的官方术语，但这样方便记忆。

上述列表不仅列出了所有限制，还描述了将静态类中的“普通”静态方法转换成扩展方法具体需要做的事情——只需添加this关键字。代码清单10-3展示了和代码清单10-1相同的类，但这一次两个方法都是扩展方法。

10

代码清单10-3 包含扩展方法的StreamUtil类

```
public static class StreamUtil
{
   const int BufferSize = 8192;

   public static void CopyTo(this Stream input, Stream output)
   {
      byte[] buffer = new byte[BufferSize];
      int read;
      while ((read = input.Read(buffer, 0, buffer.Length)) > 0)
      {
         output.Write(buffer, 0, read);
      }
   }

   public static byte[] ReadFully(this Stream input)
   {
      using (MemoryStream tempStream = new MemoryStream())
      {
         CopyTo(input, tempStream);
```

```
            return tempStream.ToArray();
        }
    }
}
```

是的，代码清单10-3唯一大的变化就是添加了两个修饰符，如加粗的部分所示。我还将方法名从Copy改成了CopyTo。马上就会看到，这个改变会使调用代码读起来更自然，虽然新的名字目前在ReadFully方法中看起来有点儿奇怪。

现在，如果我们不能使用扩展方法，那它就没什么用了。

10.2.2　调用扩展方法

前面已顺便提到了扩展方法，但还没有见识过它实际的表现。简单地说，它假装自己是另一个类型（也就是第一个方法参数的类型）的实例方法。

使用StreamUtil类的示例代码和使用工具类本身一样，都只需很小的改变。这次不是添加代码，而是移除代码。代码清单10-4是代码清单10-2的翻版，但用了“新”语法来调用CopyTo。虽然号称“新”，但实际一点都不新，它和调用实例方法的语法是一样的。

代码清单10-4　用扩展方法复制一个流

```
WebRequest request = WebRequest.Create("http://manning.com");
using (WebResponse response = request.GetResponse())
using (Stream responseStream = response.GetResponseStream())
using (FileStream output = File.Create("response.dat"))
{
    responseStream.CopyTo(output);
}
```

在代码清单10-4中，至少表面上看起来是让响应流执行复制操作。其实仍然是StreamUtil在幕后做这件事情，只是代码读起来更自然了。事实上，编译器已将CopyTo调用转换成对普通静态方法StreamUtil.CopyTo的调用。调用时，会将responseStream的值作为第一个实参的值传递（然后是output，跟平常一样）。

看到代码后，你应该能理解为什么我要将方法名从Copy更改为CopyTo了。有的名称对实例方法和静态方法都是适宜的，有的则需要调整，以获得最佳的可读性。

如果你想使StreamUtil类内的代码看起来更舒服，可以将ReadFully方法中调用CopyTo的那一行改成：

```
input.CopyTo(tempStream);
```

到此为止，对名称的修改算是各方面都满意了，虽然扩展方法完全可以作为普通的静态方法来使用，但是移植大量代码的时候扩展方法就很有用了。

你可能已经注意到，在这些方法调用中，没有任何迹象表明我们使用的是扩展方法，而不是Stream类的普通实例方法。我们可从两个角度看待这件事：如果我们的目的是使扩展方法和周围的环境尽可能地协调，尽可能少地引起恐慌，这就是一件好事；但是，如果你想快速了解真正发生的事情，这就变成一件坏事了。

在Visual Studio中，可以将鼠标对准一个方法调用，并通过出现的工具提示来了解它是不是一个扩展方法，如图10-1所示。“智能感知”（IntelliSense）也能通过方法图标或者选择某方法后的工具提示，来指明该方法是否为扩展方法。当然，你并不愿意每次都要用鼠标去对准每一个方法调用，或者小心翼翼地观察“智能感知”的显示，但大多数时候，调用的是实例方法还是扩展方法是无关紧要的。

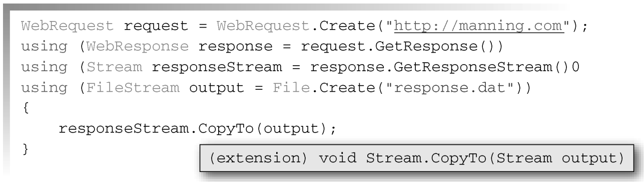


图10-1　在Visual Studio中将鼠标对准一个方法调用，即可显示该方法是否为扩展方法

关于我们的调用代码，还有一件事情是相当奇怪的——我们没有在任何地方提到`StreamUtil`！编译器当初怎么知道要使用扩展方法呢？

10.2.3　扩展方法是怎样被发现的

知道怎样调用扩展方法固然重要，但知道怎样不调用同样重要。换言之，你要知道如何实现“非请勿来”。为此，我们首先需要知道编译器怎样决定要使用的扩展方法。

如果使用`using`指令，扩展方法可以像类一样不加限制地在代码中使用。如果编译器认为一个表达式好像是要使用一个实例方法，但没有找到与这个方法调用兼容的实例方法（如不存在具有该名称的方法，或者没有重载的版本能匹配给定的实参），就会查找一个合适的扩展方法。它会检查导入的所有命名空间和当前命名空间中的所有扩展方法，并匹配那些从表达式类型到扩展类型存在着隐式转换的扩展方法。

实现细节：编译器怎样找到库中的一个扩展方法

为了决定是否使用一个扩展方法，编译器必须能区分扩展方法与某静态类中恰好具有合适签名的其他方法。为此，它会检查类和方法是否具有`System.Runtime.CompilerServices.ExtensionAttribute`这个特性，它是.NET 3.5新增的。但是，编译器不检查特性来自哪个程序集。这意味着即使你的项目面向的是.NET 2.0，仍然可以使用扩展方法——只需在正确的命名空间中使用正确的名称来定义自己的属性就可以了[①]。然后，你可以声明扩展方法，该特性会自动应用到方法和类上。编译器还会将该特性应用到包含扩展方法的程序集上，但目前在寻找扩展方法时，这还不是必需的。

在代码中引入你自己编写的系统类型的副本可能会带来一些问题，如果你以后要使用的框架版本中已经包含了这些类型的话就会出现问题。如果你确实需要使用这种技术，可以使用预处理器符号，有条件地声明特性。然后，你可以将代码分别编译成面向.NET 2.0和面向.NET 3.5（或更高）的版本。

① 即自己编写一个`System.Runtime.CompilerServices.ExtensionAttribute`类。——译者注

如果存在多个适用的扩展方法，它们可应用于不同的扩展类型（使用隐式转换），那么将使用在重载的方法中应用的“更好的转换”规则（参见9.4.4节），来选择最合适的方法。例如，假定`IChild`继承自`IParent`，而且两者都有一个同名的扩展方法，那么会优先选择`IChild`的扩展方法而不是`IParent`的。同样，该特性也用于LINQ，我们将在12.2节遇到`IQueryable<T>`接口时再来讲述。

要注意的一个重点是，如果存在适当的实例方法，则实例方法肯定会先于扩展方法使用。但是，编译器不会警告你存在一个和现有的实例方法匹配的扩展方法。例如，.NET 4为`Stream`新引入了一个方法，也叫`CopyTo`。它包含两个重载，其中一个与我们创建的扩展方法冲突。结果是新的方法总是会优先于扩展方法，因此如果在.NET 4下编译代码清单10-4，将使用`Stream.CopyTo`，而不再是`StreamUtil.copyTo`。你仍然可以使用常规的语法`StreamUtil.CopyTo(input,output)`来静态地调用`StreamUtil`方法，却永远不可能被选为扩展方法了。在本例中，这样对已有代码是没有危害的，因为新的实例方法与我们的扩展方法含义相同，使用哪个都是无所谓的。但在其他情况下，两种方法可能存在语义上的微小差别，并且在代码崩溃之前很难被发现。

扩展方法应用于代码的方式还存在一个潜在的问题——它的应用范围过于宽泛。如果同一个命名空间中的两个类含有扩展类型相同的方法，就没办法做到只用其中一个类中的扩展方法。类似地，为了通过类型的简单名称（无命名空间前缀）来使用类型，你可以导入该类型所在的命名空间，但在这样做的时候，你没有办法阻止那个命名空间中的扩展方法也被导入进来。你可能希望创建一个命名空间，仅包含定义扩展方法的静态类，以此来解决这个问题，除非该命名空间中的其他功能已经严重依赖于扩展方法（如`System.Linq`就是这种情况）。

扩展方法有一个特点，当你初次遇到它的时候，会相当惊讶，但它在某些情况下也非常有用。这个特点跟空引用有关，详情参见下一小节。

10.2.4　在空引用上调用方法

如果某人写过很多.NET程序，却从未见过因对值为空引用的变量进行方法调用而引发的`NullReferenceException`异常，那真可谓天方夜谭。在C#中，你还能在空引用上调用实例方法（不过对于非虚的调用，IL本身是支持的），但你可以在空引用上调用扩展方法。代码清单10-5对此进行了演示。注意这个例子没有采用代码段（snippet）格式，因为嵌套的类不能包含扩展方法[①]。

代码清单10-5　在空引用上调用扩展方法

```
using System;
public static class NullUtil
{
   public static bool IsNull(this object x)
   {
```

① “代码段”是作者为本书代码示例特别设计的一种格式，可直接在作者提供的Snippy程序中输入并运行。详情参见1.7节。但是，当前这个例子是一个例外，它不能用Snippy来运行，必须在Visual Studio中新建一个项目并独立运行。——译者注

```
        return x == null;
    }
}
public class Test
{
    static void Main()
    {
        object y = null;
        Console.WriteLine(y.IsNull());
        y = new object();
        Console.WriteLine(y.IsNull());
    }
}
```

代码清单10-5的输出先是True，然后是False。如果IsNull是一个普通的实例方法，Main的第2行就会抛出一个异常。但是，这里的null是IsNull的实参。在扩展方法问世前，y.Isnull()这样的写法虽然可读性更好，却不合法，只能采用NullUtil.IsNull(y)这样的写法。

在框架中，有一个特别明显的例子可以证明这种写法的好处：string.IsNullOrEmpty。在C# 3中，扩展方法可以和扩展类型的一个现有的静态方法具有相同的签名（当然，用于指定扩展类型的那个"额外"参数除外）。例如，即使string类有一个静态的、无参数的IsNullOrEmpty方法，你仍然可以创建并使用以下扩展方法：

```
public static bool IsNullOrEmpty(this string text)
{
    return string.IsNullOrEmpty(text);
}
```

从表面上看，在一个为null的变量上调用IsNullOrEmpty而不抛出异常，这似乎是一件很奇怪的事情——尤其是在你熟悉.NET 2.0的这个静态方法的前提下。但在我看来，使用扩展方法的代码更容易理解。例如，如果把表达式if (name.IsNullOrEmpty())大声地念出来，就会立即明白它的意思。

10

和往常一样，要自己多试验，看什么方案最适合你——如果你负责对别人的代码进行调试，那么要注意别人是否使用了这个技术。除非你确定一个方法不是扩展方法，否则不要想当然地认为这个方法调用会抛出异常！另外，将一个现有的方法名重用为扩展方法名时也要三思——那些只熟悉框架中静态方法的读者在遇到上述扩展方法时，有可能产生困惑。

说明 **可空性检查** 作为一个负责的开发者，你的产品方法肯定会在处理前检查参数的有效性。扩展方法的这个古怪特性很自然地会带来一个问题，即如果第一个参数为空（假设并不希望这样），应该抛出什么样的异常呢？是ArgumentNullException吗，就像它是一个普通的参数？还是NullReferenceException呢？如果扩展方法为实例方法，会抛出这样的异常吗？我建议抛出前者，因为尽管从扩展方法的语法上看并不十分明显，但它（第一个参数）确实只是一个参数。这也是微软在框架中为扩展方法所采取的处理措施，因此这样做也能带来一致性的好处。最后，要记住扩展方法仍然可以像普通的静态方法那样调用。这时，ArgumentNullException显然应该是首选。

在知道了扩展方法的语法和行为之后，接着让我们看看它们的一些例子。这些扩展方法是.NET 3.5框架的一部分。

10.3　.NET 3.5 中的扩展方法

在框架中，扩展方法最大的用途就是为LINQ服务。有的LINQ提供器包含了几个供辅助的扩展方法，但有两个类特别醒目，Enumerable和Queryable，两者都在System.Linq命名空间中。在这两个类中，含有许许多多的扩展方法：Enumerable的大多数扩展的是IEnumerable<T>，Queryable的大多数扩展的是IQueryable<T>。IQueryable<T>的作用将在第12章讲述，目前让我们将重点放在Enumerable上。

10.3.1　从**Enumerable**开始起步

即使只是稍微浏览一下Enumerable，你和LINQ之间的距离也会变得非常之近。事实上，你大多数时候并不需要功能强大的查询表达式来解决某个问题。Enumerable含有大量方法，本节的目的并不是要讲述所有这些方法，而是让你对它们有足够多的认识，然后自己去试验。摆弄Enumerable提供的一切真的是一件乐事——但就目前来说，你最好还是启动Visual Studio或者LINQPad来进行试验（而不是使用我提供的Snippy程序），因为智能感知可以为你的操作提供相当大的帮助。附录A简单介绍了Enumerable的这些方法的行为。

本节所有完整的例子处理的都是一个简单的情况：从一个整数集合开始，以不同的方式转换它。实际情况则可能要更复杂一些，你可能需要处理与业务有关的类型（business-related type）。本节末尾会提供两个例子，它们只演示了在实际工作环境中需要进行转换的内容，完整源代码可从本书网站获取。虽然这两个例子比较简单，但仍然要比一个单纯的数字集合难一些。

在这个过程中，你可以考虑一下自己最近做过的项目，想一想是否有些地方可以利用这里描述的操作使代码变得更简单、更易读。

在Enumerable中，有几个方法不是扩展方法，本章我们会用到其中一个。Range方法获取两个int参数：一个起始数，一个是要生成的结果的数目。结果是一个IEnumerable<int>，显然，它每次返回一个数字。

为了演示Range方法，并提供一个供操作的框架，我们假定要打印数字0~9，如代码清单10-6所示。

代码清单10-6　用Enumerable.Range打印数字0~9

```
var collection = Enumerable.Range(0, 10);

foreach (var element in collection)
{
   Console.WriteLine(element);
}
```

代码清单10-6中没有调用扩展方法，就是一个普通的静态方法。而且这段代码确实只是打印

数字0~9——我从来没有说过这段代码会有什么惊人之举。

说明　延迟执行　Range方法并不会真的构造含有适当数字的列表，它只是在恰当的时间生成那些数。换言之，构造的可枚举的实例并不会做大部分工作。它只是将东西准备好，使数据能在适当的位置以一种“just-in-time”的方式提供。这称为延迟执行，是LINQ的一个核心部分。我们已经在第6章讲解迭代器块的时候看到过这种行为了，下一章将更多地讨论这方面的内容。

面对一个数字序列（已排好序），我们能做的最简单的事情就是反转它。代码清单10-7用Reverse扩展方法来做这件事情——它返回一个IEnumerable<T>，这个IEnumerable<T>会生成与原始序列相同的元素，只是采用相反的顺序。

代码清单10-7　用Reverse方法来反转一个集合

```
var collection = Enumerable.Range(0, 10)
                           .Reverse();

foreach (var element in collection)
{
   Console.WriteLine(element);
}
```

我们可以预测到上述代码会打印9,8,7,……,0。我们（表面上）调用IEnumerable <int>的Reverse方法，并返回相同的类型。这种基于一个可枚举实例返回另一个可枚举实例的编程模式在Enumerable类中是随处可见的。

效率问题：缓冲和流式技术

框架提供的扩展方法会尽量尝试对数据进行“流式”（stream）或者说“管道”（pipe）传输。要求一个迭代器提供下一个元素时，它通常会从它链接的迭代器获取一个元素，处理那个元素，再返回符合要求的结果，而不用占用自己更多的存储空间。执行简单的转换和过滤操作时，这样做非常简单，可用的数据处理起来也非常高效。但是，对于某些操作来说，比如反转或排序，就要求所有数据都处于可用状态，所以需要加载所有数据到内存来执行批处理。缓冲和管道传输方式，这两者的差别很像是加载整个DataSet读取数据和用一个DataReader来每次处理一条记录的差别。使用LINQ时务必想好真正需要的是什么，一个简单的方法调用可能会严重影响性能。

流式传输（streaming）也叫惰性求值（lazy evaluation），缓冲传输（bufferring）也叫热情求值（eager evaluation）。例如，Reverse方法使用了延迟执行（deferred execution）[①]，它在第

① 惰性求值和热情求值都属于延迟执行的求值方式，与立即执行（immediately execution）相对。Stack Overflow上的一个帖子很好地阐述了它们之间的区别（参见http://stackoverflow.com/questions/2515796/deferred-execution-and-eager-evaluation）。——译者注

一次调用MoveNext之前不做任何事情。但随后却热切地（eagerly）对数据源求值。我个人并不喜欢惰性和热情这种术语叫法，不过仁者见仁智者见智（参见http://mng.bz/3LLM）。

下面让我们做一些更刺激的事情——我们将用一个Lambda表达式来删除偶数。

10.3.2　用**Where**过滤并将方法调用链接到一起

Where扩展方法是对集合进行过滤的一种简单但又十分强大的方式：它接受一个谓词，并将其应用于原始集合中的每个元素。Where同样返回一个IEnumerable<T>，但这一次结果集合中只包含与谓词匹配的元素。

代码清单10-8对此进行了演示，它先向整数集合应用奇偶过滤器，再对其进行反转。这里不一定要使用Lambda表达式——例如，可以使用早先创建的委托，或者使用匿名方法。在本例以及其他许多现实的情况中，将过滤逻辑内联可以更加简单，Lambda表达式可以使代码保持整洁。

代码清单10-8　用Lambda表达式作为Where方法的参数，从而只保留奇数

```
var collection = Enumerable.Range(0, 10)
                           .Where(x => x % 2 != 0)
                           .Reverse();
foreach (var element in collection)
{
   Console.WriteLine(element);
}
```

代码清单10-8打印数字9,7,5,3和1。希望你此时已经注意到了一个模式——我们将方法调用链接到一起了。链接并不是一个新概念。例如，StringBuilder.Replace总是返回你调用的那个实例，所以以下代码是合法的：

```
builder = builder.Replace("<", "&lt;")
                 .Replace(">", "&gt;")
                 ...
```

相对之下，String.Replace返回一个字符串，但每次都是一个新字符串——这也可以链接，但是方式略有不同。这两种模式都很常用。“返回相同的引用”模式用于易变类型，而“返回新实例（该实例为原始实例更改后的副本）”模式则用于不易变类型。

以上两种模式都可轻松用于实例方法，而扩展方法允许将静态方法调用链接到一起。这其实是扩展方法存在的主要原因之一。虽然它们对其他工具类来说也是有用的，但它们真正强大的地方还是在于能够以一种自然的方式将静态方法链接起来。这正是.NET 3.5的扩展方法主要出现在Enumerable和Queryable中的原因：LINQ针对数据处理进行了专门的调整，将各个单独的操作链接成一条管道，然后让信息在这个管道中传输。

如果Reverse和Where不是扩展方法，那么，代码清单10-8的第一部分可以换用两种显而易见的方式来编写。一种方式是使用一个临时变量，它能保持结构原封不动：

```
var collection = Enumerable.Range(0, 10);
collection     = Enumerable.Where(collection, x => x % 2 != 0);
collection     = Enumerable.Reverse(collection);
```

说明 效率问题：重新排序方法调用以避免浪费 虽然我不喜欢没有很好的理由就去进行一些微优化[1]，但仍有必要看看代码清单10-8中的方法调用的顺序。本来是可以在Reverse调用之后进行Where调用，这样的结果是相同的。然而，那样做会造成浪费——Reverse调用必须计算出偶数在序列中的位置，而偶数在最终的结果中是会被丢弃的。对本例来说，这当然不会造成太大的差别，但在处理大量数据时，性能就可能受到严重影响。如果能在不影响可读性的前提下减少不必要的工作，那就是一件好事情！那并不是说你一定要在管道的起始处布置过滤器（Where），只是说你需要慎重考虑任何重新排序操作，确定重新排序后仍然得到正确的结果。

显然，代码的含义远不及代码清单10-8表达得清楚。但如果换用第二种保持“单语句”风格的方式，那么结果会更糟：

```
var collection = Enumerable.Reverse
                   (Enumerable.Where
                      (Enumerable.Range(0, 10),
                         x => x % 2 != 0));
```

方法调用的顺序看起来是反的，因为最里边的方法调用（Range）会先执行，然后再执行其他的，执行顺序是从内向外的。尽管只有三个方法调用，但还是异常丑陋。甚至比使用了更多操作符的查询还要糟糕。

在我们继续后面的内容之前，先来看看Where方法都做了些什么。

10.3.3 插曲：似曾相识的Where方法

你肯定不会对Where方法感到陌生，因为我们在第6章曾经实现过。我们需要做的是将代码清单6-9的方法转换为扩展方法，并将委托类型由Predicate<T>改为Func<T, bool>，这样我们就实现了一个非常完美的Enumerable.Where方法的替代品。

```
public static IEnumerable<T> Where<T>(this  IEnumerable<T> source,
                                      Func<T, bool> predicate)
{
   if (source == null || predicate == null)
   {
      throw new ArgumentNullException();
   }
   return WhereImpl(source, predicate);
}
private static IEnumerable<T> WhereImpl<T>(IEnumerable<T> source,
                                           Func<T, bool>  predicate)
{
   foreach (T item in source)
   {
      if (predicate(item))
      {
```

[1] 即Micro-optimization，相对于结构优化而言。——译者注

```
            yield return item;
        }
    }
}
```

我们也可以转换代码清单6-9示例中后一部分的调用，使其与LineReader类链接起来：

```
foreach (string line in LineReader.ReadLines("../../FakeLinq.cs")
                                  .Where(line => line.StartsWith("using")))
{
   Console.WriteLine(line);
}
```

这是一个未使用System.Linq命名空间的有效的LINQ查询。如果你声明了一个适当的Func委托和[ExtensionAttribute]，它甚至可以在.NET 2.0下正常工作。你甚至可以在一个查询表达式中将该实现作为其where子句（同样是面向.NET 2.0），我们将在下一章介绍这一内容，现在还是不要超前了。

过滤是查询中最简单的操作之一，还有就是转换（或者说投影）结果。

10.3.4　用Select方法和匿名类型进行投影

Enumerable中最重要的投影方法就是Select——它操纵一个IEnumerable<TSource>，把它投影成一个IEnumerable<TResult>。具体的投影是通过一个Func<TSource, TResult>来完成的，它代表要在每个元素上执行的转换，采用的是委托的形式。Select和List<T>中的ConvertAll方法很像，但它能操纵任意可枚举的集合。它利用了“延迟执行”技术，只有在每个元素被请求的时候才真正执行投影。

当初介绍匿名类型时，我说过它们和Lambda表达式以及LINQ表达式一起使用是最有用的——现在给出的这个例子能很好地说明这一点。我们现在已经得到了0~9的奇数（逆序）——接着让我们创建一个类型来封装数字的平方根以及原始数字。代码清单10-9展示了投影过程，并用一种稍有变化的方式输出结果。我还调整了空白，这纯粹是为了满足书本印刷的需要。

代码清单10-9　用Lambda表达式和匿名类型进行投影

```
var collection = Enumerable.Range(0, 10)
       .Where(x => x % 2 != 0)
       .Reverse()
       .Select(x => new { Original = x, SquareRoot = Math.Sqrt(x) } );

foreach (var element in collection)
{
   Console.WriteLine("sqrt({0})={1}",
                     element.Original,
                     element.SquareRoot);
}
```

这一次的Collection的类型不是IEnumerable<int>，而是IEnumerable<Something>，其中Something是由编译器创建的匿名类型。我们不能显式地指定集合变量的类型，除非指定成非泛型的IEnumerable类型或object。正是因为使用了隐式类型（var），所以我们才能在输出

结果时使用Original和SquareRoot属性。

代码清单10-9的输出如下：

```
sqrt(9)=3
sqrt(7)=2.64575131106459
sqrt(5)=2.23606797749979
sqrt(3)=1.73205080756888
sqrt(1)=1
```

当然，Select方法不一定非得使用一个匿名类型——我们可以只选择数字的平方根，而丢弃原始值。这时的结果为IEnumerable<double>类型。除此之外，还可以手动写一个类型来封装整数及其平方根——只是在当前这种情况下，使用匿名类型是最容易的罢了。

下面让我们研究最后一个方法，从而圆满结束对Enumerable的讨论：OrderBy。

10.3.5 用**OrderBy**方法进行排序

对数据排序是处理数据时的一项常规要求。在LINQ中，这一般是通过OrderBy或OrderByDescending方法来实现的。如果需要根据数据的多个属性排序，那么后面还可以跟随ThenBy或ThenByDescending。基于多项属性排序这种复杂的操作往往会用到复杂的比较，但如果能以一系列简单的比较来定义这种排序，那么通常都能化繁为简。

为了对此进行演示，我打算稍微改变一下要使用的操作。我们先构造−5~5的整数集合（共包括11个整数），然后投影到一个包含原始数字及其平方（而不是平方根）的匿名类型。最后，我们先按平方进行排序，再按原始数字进行排序。代码清单10-10展示了所有这些操作。

代码清单10-10 根据两个属性对序列进行排序

```
var collection = Enumerable.Range(-5, 11)
      .Select(x => new { Original = x, Square = x * x })
      .OrderBy(x => x.Square)
      .ThenBy(x => x.Original);

foreach (var element in collection)
{
   Console.WriteLine(element);
}
```


注意，除了对Enumerable.Range的调用，整段代码读起来就像“自然语言”一样。我们用匿名类型的ToString实现来进行格式化，下面是结果：

```
{ Original = 0, Square = 0 }
{ Original = -1, Square = 1 }
{ Original = 1, Square = 1 }
{ Original = -2, Square = 4 }
{ Original = 2, Square = 4 }
{ Original = -3, Square = 9 }
{ Original = 3, Square = 9 }
{ Original = -4, Square = 16 }
{ Original = 4, Square = 16 }
{ Original = -5, Square = 25 }
{ Original = 5, Square = 25 }
```

正如我们期待的那样，“主”排序属性是Square，但对于平方值相同的两个值，负的原始值总是排在正的原始值前面。而如果用单一的比较来实现相同的目标（一般情况下，对于本例这类特殊的问题可以用专门的数学技巧来处理）则要复杂得多——会复杂到你不想在Lambda表达式中“内联”代码的程度。

需要注意的是，排序不会改变原有集合——它返回的是新的序列，所产生的数据与输入序列相同，当然除了顺序。这与List<T>.Sort和Array.Sort不同，它们会改变列表或数组中元素的顺序。LINQ操作符是无副作用的：它们不会影响输入，也不会改变环境。除非你迭代的是一个自然状态序列（如从网络流中读取数据）或使用含有副作用的委托参数。这是函数式编程的方法，可以使代码更加可读、可测、可组合、可预测、健壮并且线程安全。

我们只是展示了Enumerable的大量扩展方法中的少数几个，希望你已经体会到了它们是如何优雅、简洁地链接到一起。下一章将看到如何使用C# 3的新语法（查询表达式）来表达这些操作。另外，届时还会讲到这里没有提及的其他一些操作。但请记住，你并非一定要使用查询表达式——通常，更简单的做法是调用Enumerable中的几个方法，用扩展方法将操作链接到一起。

在见识了所有这些如何应用于我们的“数字集合”例子之后，接着是时候履行我们的承诺，展示一些与实际工作有关的例子了。

10.3.6　涉及链接的实际例子

作为开发者，我们做的许多工作就是来回移动数据。事实上，对于许多应用程序，这是唯一有意义的事情。一些单独存在的用户界面、Web服务、数据库和其他组件都是将数据从一个地方移动到另一个地方，或者从一种形式转换成另一种形式。本节讨论的扩展方法可以很好地解决许多实际问题，对此我们丝毫不应感到惊讶。

下面只打算展示两个例子，相信你能以它们为跳板，更加深入地思考自己的业务需求，以及如何利用C# 3和Enumerable类，以一种更具表达力的方式来解决自己的问题。每个例子只包含一个示例查询，这足以帮助你理解代码的用途，同时不必纠缠于一些细枝末节。本书网站上有完整的工作代码。

1. 聚合：工资汇总

第一个例子涉及由几个部门组成的一家公司。每个部门都有大量员工，每个员工都有一份工资。假定要对各部门的总工资制作报表，工资最高的部门排在前面。查询很简单：

```
company.Departments
       .Select(dept => new
       {
          dept.Name,
          Cost = dept.Employees.Sum(person => person.Salary)
       })
       .OrderByDescending(deptWithCost => deptWithCost.Cost);
```

这个查询用一个匿名类型来记录部门名称（用一个投影初始化器）以及该部门的所有员工的总工资。工资汇总使用的是一个含义明确的Sum扩展方法，它同样来自Enumerable。

在结果中，部门名称和总工资可以以属性的方式来获取。如果想使用原始的部门引用，只需

更改Select方法中使用的匿名类型就可以了。

2. 分组：统计分配给开发者的bug数量

如果你是一名专业开发者，那么必定知道许多项目管理工具都会提供不同的度量标准（metric）。如果能访问到原始数据，LINQ就能将其转换为任何形式。

作为一个简单的例子，下面让我们来看一个开发者列表，并统计当前分配给他们的bug数量：

```
bugs.GroupBy(bug => bug.AssignedTo)
    .Select(list => new { Developer = list.Key, Count = list.Count() })
    .OrderByDescending(x => x.Count);
```

这个查询使用了GroupBy扩展方法，它按一个投影（本例是分配来修正这个bug的开发者）对原始集合进行分组，结果是一个IGrouping<TKey,TElement>。GroupBy有多个重载版本，这里使用的是最简单的。然后选择键（开发者的姓名）和分配给他们的bug的数量。之后，我们对结果进行排序，最先显示分配到bug数量最多的开发者。

研究Enumerable类时，往往会感觉搞不清楚具体发生的事情——例如，GroupBy的一个重载版本居然有4个类型参数和5个“普通”参数（3个是委托）。但是，不要惊慌——只要按照第3章描述的步骤慢慢梳理，将不同的类型赋给不同的类型参数，直到清楚呈现出方法的样子。这样，理解起来就容易多了。

这些例子不是具体针对某个方法调用，但我希望你能体会到将方法调用链接起来之后所发挥的巨大作用。在这个链条中，每个方法都获取一个原始集合，并以某种形式返回另一个原始集合——中间可能过滤掉一些值，可能对它们进行排序，可能转换每一个元素，可能聚合某些值，或者做其他处理。在许多情况下，最终的代码都易读、易懂。在其他情况下，它最起码也会比使用以前版本的C#写的等价代码简单得多。

下一章研究查询表达式时，我们会使用缺陷跟踪[①]（Defect Tracking）的例子作为我们的示例数据。在了解了C#提供的一些扩展方法之后，接着让我们探讨一下如何自己写扩展方法，以及在什么情况下写才能有意义。

10.4 使用思路和原则

类似于隐式类型的局部变量，扩展方法也是有争议的。许多时候，虽然很难说它们是否会使代码的总体目标变得更难理解，但与此同时，它们确实有可能让人弄不明白调用的是什么方法。用我的一个大学老师的话来说：“我隐藏真相是为了让你看到更大的真相。”如果你觉得代码最重要的就是它的结果，扩展方法肯定相当合你的胃口！如果觉得实现更重要，那么显式调用静态方法就显得更清晰。实际上，这是“是什么”（what）和“怎么做”（how）之间的差异。

我们已经讨论了如何将扩展方法用于工具类和方法链接，在进一步讨论它的优缺点之前，有必要先明确可能不是很明了的两个方面的问题。

① 缺陷跟踪：软件开发中的一个重要环节，旨在预防和减少缺陷，提高软件开发能力。——译者注

10.4.1　“扩展世界”和使接口更丰富

C#编译器团队的前开发人员Wes Dyer在他的博客（参见http://blogs.msdn.com/b/wesdyer/）中就C#的方方面面进行了精彩的诠释。有一篇关于扩展方法的文章（参见http://mng.bz/I4F2）特别引人注目。文章标题是“Extending the World”（扩展世界），其中描述了扩展方法如何通过调整环境来满足你的需求，从而使代码变得更易读：

> 对于一个给定的问题，程序员通常习惯于构建一个解决方案，直到最终能满足需求。现在，我们可以扩展世界来迎合解决方案，而不是一直构建方案，直到最终满足需求。如果一个库没有提供你需要的，就扩展这个库来满足你的需求。

这段话的立意要高出单纯使用一个工具类时的情况。通常，开发者只是在看到相同形式的代码在多个地方重复出现时，才会着手创建工具类。但是，对库进行扩展是为了明确代码的表达方式，从而尽可能地避免重复。扩展方法可以使调用代码“觉得”库中资源要比实际的“丰富”。

我们已在`IEnumerable<T>`上体验到了这一点，即使最简单的实现，似乎也支持大量操作，比如排序、分组、投影和过滤。当然，好处并非仅限于接口。你还可以通过枚举、抽象类等来“扩展世界”。

.NET Framework有一个很好的例子演示了扩展方法的另一个用途：流畅接口。

10.4.2　流畅接口

过去英国有一个电视节目叫做Catchphrase（名言）。节目内容是，参赛者将看一个屏幕，屏幕上的一段动画会演示具有一定含义的短语或俗语，参赛者通过这些猜测其真正的含义。主持人为了帮助参赛者，通常会提示他们：“说出你看到的。”这与流畅接口的背后理念十分相似——如果把代码逐字逐句地读出来，它的作用应该跃然“屏”上，好像本来就是用自然语言写成的一样。这个术语最早是由Martin Fowler（参见http://mng.bz/3T9T）和Eric Evans提出来的。

如果你熟悉DSL[①]（Domain Specific Language，领域特有语言），可能会奇怪流畅接口和DSL有什么区别。人们就这个主题写了许多文章，但大家一致认为DSL有更大的自由来创建它自己的句法和语法，而流畅接口受限于“宿主”语言（本例中就是C#了）。

在框架中，流畅接口的一个很好的例子就是`OrderBy`和`ThenBy`方法：用Lambda表达式稍加诠释，代码准确地描述了它要做的事情。在前面代码清单10-10的数字例子中，可以这样读代码：“Order by平方，Then by原始数。”含义一目了然。这样的语句能像完整的英文句子那样读，而不是由独立的“名词动词化”的短语构成。

写流畅接口时，可能要求思维方式发生改变。方法名有违传统的“描述性动词”形式，像“And”、“Then”和“If”这样的词有时适合作为流畅接口中的方法名。方法本身一般只是为将来

① 关于DSL（domain-specific language）的详细内容可以参考维基百科：http://en.wikipedia.org/wiki/Domain-specific_language。——译者注

的调用建立一个上下文。对于它们返回的类型来说，通常唯一的作用就是充当调用之间的一座桥梁。图10-2展示了这种“桥接”是如何进行的。它只使用了两个方法（分别扩展`int`和`TimeSpan`），却使可读性发生了翻天覆地的变化。

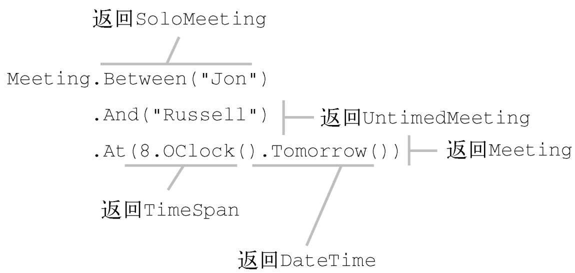


图10-2　分解一个用来创建会议的流畅接口表达式，使用扩展方法从`int`创建`TimeSpan`，再创建`DateTime`，以此来确定会议时间

这个例子的语法可以有许多不同的形式。例如，你可以为一个`UntimedMeeting`（未确定时间的会议）添加更多的与会者，或者在指定与会者之前创建在一个特定时间召开的`Unattended-Meeting`（未确定与会者的会议）。如果要了解更多关于DSL的内容，可以参考Ayende Rahien的*DSLs in Boo: Domain-Specific Languages in .NET*（Manning,2010）一书。

C# 3只支持扩展方法而不支持扩展属性，这多少限制了流畅接口。这意味着我们不能写出下面这样的表达式：1.week.from.now或者2.days + 10.hours（在Groovy中，这些都是有效的，安装一个合适的组件就可以，参见http://groovy.codehaus.org/Google+Data+ Support）。但是，通过添加几个冗余的圆括号，我们可以获得类似的结果。虽然在一个数字上调用方法看起来很奇怪，比如`2.Dollars()`或者`3.Meters ()`，但不得不承认它们的意思确实变清晰了。如果没有扩展方法，要对不受自己控制的类型（如数字）执行这种简洁清晰的操作，是不可能的。

写作本书时，开发社区仍在对流畅接口持观望态度。尽管许多模拟库和单元测试①已经具备流畅的意思，但在大多数领域，还是极少用到它们。虽然普适性差一些，但在合适的情况下，它们确实会从根本上改变调用代码的可读性。例如，使用我的MiscUtil库中适当的扩展方法，可以迭代我过去生活的每一天，并且非常具有可读性：

```
foreach (DateTime day in 19.June(1976).To(DateTime.Today)
                                      .Step(1.Days()))
```

尽管与时间范围相关的实现细节异常复杂，但扩展方法`19.June(1976)`和`1.Days()`还是十分简单的。这是特定区域的代码，你可能不会用在生产代码中，但它可以使单元测试变得更加轻松。

当然，这些并不是扩展方法唯一的用途。我已经将它们用于参数验证、实现LINQ的替代方法、向LINQ to Objects添加自定义操作符、简化复合比较的构建、向枚举添加更多标记相关的功能，等等。我个人就常常惊叹于在使用得当的前提下，如此简单的一个小特性就能对可读性产生

① 进行单元测试时，为了避免纠缠于不必要的细节，可以用一些对象来模拟真实对象，这些对象称为模拟对象，相应的库就可以称为模拟库。——译者注

深远的影响。注意这里的关键词是“得当”，这说起来容易做起来难。

10.4.3　理智使用扩展方法

我无权对你如何编写代码指手画脚。也许能写一些测试来客观地评价代码对于“平均水平”的开发者而言的可读性如何，但只有那些需要使用和维护你的代码的人才会关注可读性。所以，要尽量同相关人员协商。当然，这要取决于项目的类型及其用户。但是，如果能给出不同的选择，并从需要读代码的人那里获得相应的反馈，应该是不错的。在许多情况下，扩展方法都能轻松地实现这一点。在能够实际工作的代码中，你可以同时演示两种方案——将方法转换成扩展方法无碍你像往常一样以相同的方式显式调用它。

要问的主要问题是我在本节开头提出的：代码“做什么”是否比“怎么做”更重要？对于不同的人或在不同的情况下对这个问题的回答都是不同的。但是，有一些原则的东西是需要记住的。

- 开发团队中的每个人都应该理解扩展方法，并且知道在什么地方可以使用它们。要尽可能地避免让代码维护人员感到“突兀”。
- 将扩展方法放到它们自己的命名空间，可有效防止被误用。这样一来，即使读代码的时候不是那么一目了然，至少写它的开发者会注意到他正在做的事情。用项目或公司那一级的名称来定义这个命名空间的名称。甚至可以更进一步，为每个被扩展类型都使用一个单独的命名空间。例如，可以为扩展了`System.Type`的类创建一个`TypeExtensions`命名空间。
- 在扩展广泛使用的类型（如数字、`object`等）之前，或编写扩展类型实际为类型参数这样的扩展方法之前，要深思熟虑。有些指导书会尽可能地建议你不要这样做。而我认为这样的扩展方法可以存在，但必须值得存在。在这种情况下，将扩展方法设计为内部的或放置于自己的命名空间中，这一点就显得更加重要：我可不希望在每次使用整数时，智能感知都会提示`June`这个扩展方法。只有在那些使用了与日期和时间相关的扩展方法的类中，我才希望这样。
- 写扩展方法应该始终是一个有意识的决定，不要把它培养成一个习惯。绝对不是每个静态方法都该变成一个扩展方法。
- 在文档中指出第一个参数（在该值上调用扩展方法）是否允许为`null`——如果不允许，就在方法中检查值，并在必要的时候抛出一个异常（`Argument Null Exception`）。
- 注意，如果方法名已经在扩展类型中使用，就不要再使用这个名称。如果扩展类型是框架中的类型，或者来自某个第三方库，请在库的版本发生改变时检查自己的所有扩展方法。也许你很幸运，新旧方法的含义完全相同（就像`Stream.CopyTo`一样）。但即便如此，你还是应该删除这个扩展方法。
- 想一想它们是否影响了自己的编程效率。和隐式类型一样，强迫自己使用一个不喜欢的功能是没有多大意义的。
- 将应用于同一个扩展类型的扩展方法分组到一个静态类中。有的时候，相关的类（比如`DateTime`和`TimeSpan`）的扩展方法可以分组到一起。但是，如果扩展方法作用于迥然不同的类型（比如`Stream`和`string`），就不要把它们分组到同一个类中了。

- 在两个不同的命名空间中添加名字相同、扩展类型也相同的两个扩展方法时一定要三思，尤其是在两个方法都适用（它们有相同数量的参数）的情况下。合理的做法是添加或删除一个using指令，就可以使程序构建失败。但是，即使添加或删除一个using指令，程序也能构建，只是行为可能已经发生了变化[①]，这样就比较烦人了。

以上原则很少有特别明确的。从某种程度上说，你必须摸着石头过河，自己探索在什么情况下该使用或放弃扩展方法。如果你永远不写自己的扩展方法，只是使用与LINQ有关的扩展方法来增强可读性，也是完全合情合理的。不过，你至少应该思考一下假如使用自己的扩展方法，会带来哪些可能性。

10.5 小结

扩展方法的原理一点儿都不复杂——它只是一个简单的特性，很容易使用和演示。但是，它们的优点（和相应的代价）则很难以一种明确的方式讲述出来。换言之，它是一个必须自己去体验的东西，不同的人会对它的价值有不同的理解。

在本章中，我尝试把所有内容都讲到了一点——在介绍框架提供的扩展方法之前，我先对它们在语言中想要达到的目标进行了一番阐述。从某些方面说，本章是LINQ的非常浅显的入门章：下一章深入探讨查询表达式的时候，我们会重温一些已经见到过的扩展方法，另外还会介绍一些新的。

Enumerable类中有大量方法，我们接触到的只是冰山一角。非常有趣的一个办法是自己设想一个情形（不管是虚构的还是真实项目中有的），然后浏览MSDN来找到可以帮助自己的内容。我建议你使用一些描述的一个沙盒[②]项目来摆弄提供的扩展方法，这感觉像是游戏，而不是工作。我们学东西不应过于急功近利——每次都是为了解决当前的一个实际问题才去学习。本书附录A中列出了LINQ的一些标准查询操作符，其中包含了Enumerable中的许多方法。

在软件工程领域，新的模式和实践准则层出不穷，来自一些系统的设计思想经常会“流窜”到另一些系统中。这其实是让软件开发始终保持新鲜感的原因之一。扩展方法允许采取以前C#不支持的方式来写代码，创建流畅接口，以及改变环境来迎合代码，而不是采用相反的方式。本章讨论的这些技术确保了可以使用新的C#特性来进行更有趣的开发，它们既可以单独使用，也可以组合起来使用。

但是，革新明显不会止步于此。如果只是少数几个调用，扩展方法确实不错。在下一章中，我们要讨论两个真正威力无穷的工具：查询表达式和全功能的LINQ。

① 因为可能选择的并不是你希望的那个扩展方法。——译者注

② 沙盒是运行未测试代码或非信任代码的一种安全机制，详见http://en.wikipedia.org/wiki/Sandbox_(computer_ security)。——译者注

第 11 章

查询表达式和LINQ to Objects

本章内容

- ❑ 流式处理数据和延迟执行序列
- ❑ 标准查询操作符和查询表达式转换
- ❑ 范围变量和透明标识符
- ❑ 投影、过滤和排序
- ❑ 连接和分组
- ❑ 选择要使用的语法

现在，你可能已经对那些围绕LINQ的增强功能感到厌烦了吧。我们已经在前面看到了一些例子，你肯定也在网上见过了一些例子和文章。不过，我们要将“神话”同现实区分开：

- ❑ LINQ不能把非常复杂的查询转换为一行代码；
- ❑ 使用LINQ不意味着你从此不再需要面对原生的SQL；
- ❑ LINQ不可能魔法般地让你成为架构天才。

不过话说回来，在面向对象的开发环境中，LINQ依然是我所见到的最好的查询表达方式。它虽然不是银子弹，却是你的开发“兵器库”中一个非常强大的工具。我们将分析LINQ两个截然不同的方面：框架支持和查询表达式的编译器转换。后者一开始看上去有点古怪，不过我保证你一定会爱上它的。

查询表达式实际上是由编译器“预处理”为“普通”的C# 3代码，接着以完全普通的方式进行编译。这种巧妙的方式，把查询集成到了语言当中，而无须把语义改得乱七八糟的。本章的大部分内容就是罗列出由编译器完成的预处理转换，以及展示使用`Enumerable`扩展方法时取得的效果。

本章你不会看到任何SQL或XML——所有这些都留到第12章。不过，有了本章作为基础，在我们讲到LINQ提供器的时候，你就能理解它做了哪些更令人振奋的事情。可以认为我是个扫兴的人，因为我还是打算去除一些它的魔幻色彩。但即使没有这些神秘的光环，LINQ依旧还是非常酷的。

首先，让我们来想一下LINQ究竟是什么，以及我们将要如何分析它。

11.1 LINQ 介绍

面对LINQ这样一个庞大的主题，在准备实际研究它之前，需要一定的知识储备。在本节中，我们会了解一下LINQ背后的几个核心原则，以及本章和下一章中所有例子要用到的数据模型。我知道你迫不急待地想马上深入代码，所以我不会长篇大论。

11.1.1 LINQ中的基础概念

本章的大部分内容都是致力于解释C# 3编译器如何处理查询表达式，不过在我们很好地理解LINQ的整个底层思想之前，没有多大意义。降低两种数据模型之间的阻抗失配的过程中，遇到的一个问题就是，通常会涉及创建另外一个模型来作为桥梁。本节用LINQ最重要的一个方面——序列（sequence）——来开始描述LINQ模型。

1. 序列

你当然应该对序列这个概念感觉很熟悉：它通过`IEnumerable`和`IEnumerable<T>`接口进行封装，当我们在第6章学习迭代器的时候，已经深入地研究了它。序列就像数据项的传送带——你每次只能获取它们一个，直到你不再想获取数据，或者序列中没有数据了。

序列和其他集合数据结构（比如列表和数组）之间最大的区别就是，当你从序列读取数据的时候，通常不知道还有多少数据项等待读取，或者不能访问任意的数据项——只能是当前的这个。实际上，一些序列永远不会结束：例如，你能轻易地拥有一个随机数的无限序列。列表和数组也能作为序列，因为`List<T>`实现了`IEnumerable<T>`——不过，反过来并不总是可行。比如，你不能拥有一个无限的数组或列表。

序列是LINQ的基础。在你看到一个查询表达式的时候，应该要想到它所涉及的序列：一开始总是存在至少一个序列，且通常在中间过程会转换为其他序列，也可能和更多的序列连接在一起。网上的LINQ查询例子往往没有什么解释，这是因为拆分为序列依次去看的话，就很有说明性了。同样，它也有助于阅读代码和编写代码。用序列思考需要智慧，甚至有时需要点跳跃思维，但如果能够做到，在使用LINQ的时候，这对于你的帮助是不可估量的。

来看一个简单的例子，我们在人员列表上执行另外一个查询表达式。和之前一样，我们应用同样的转换，不过附加了一个过滤器，来保证只有成年人出现在结果序列中：

```
var adultNames = from person in people
                 where person.Age >= 18
                 select person.Name;
```

图11-1以图表形式把这个查询表达式分解成了独立步骤。

每一个箭头代表一个序列——描述在左边，示例数据在右边。每个框都代表查询表达式的一个步骤。最初，我们具有整个家庭成员（用`Person`对象表示）。接着经过过滤后，序列就只包含成人了（还是用`Person`对象表示）。而最终的结果以字符串形式包含这些成人的名字。每个步骤就是得到一个序列，在序列上应用操作以生成新的序列。结果不是字符串`"Holly"`和`"Jon"`——

11

而是`IEnumerable <String>`，这样，在从里面一个接一个获取元素的时候，将首先生成`"Holly"`，其次得到`"Jon"`。

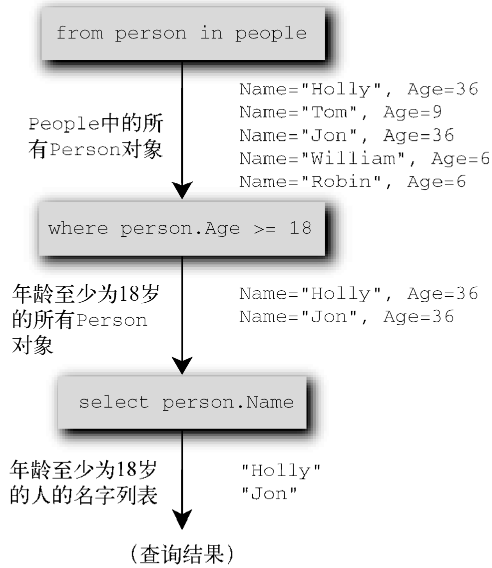


图11-1　将一个简单的查询表达式分解为相关的序列和转换

刚开始，这个例子还算简单，不过后面我们会将同样的技术应用于更复杂的查询表达式，以便能更容易地理解它们。某些高级的操作涉及多个输入序列，但比起一下子理解整个查询，按照步骤分解还是要容易得多。

那么，序列为什么如此重要？这是由于，它们是数据处理的流模型的基础，让我们能够只在需要的时候才对数据进行获取和处理。

2. 延迟执行和流处理

显示在图11-1中的查询表达式被创建的时候，不会处理任何数据，也不会访问原始的人员列表也未被访问[①]。而是在内存中生成了这个查询的表现形式。判断是否为成人的谓词，以及人到人名的转换，都是通过委托实例来表示的。只有在访问结果`IEnumerable<string>`的第一个元素的时候，整个车轮才开始向前滚动。

LINQ的这个特点称为延迟执行。在最终结果的第一个元素被访问的时候，`Select`转换才会为它的第一个元素调用`Where`转换。而`Where`转换会访问列表中的第一个元素，检查这个谓词是否匹配（在这个例子中，是匹配的），并把这个元素返回给`Select`。最后，依次提取出名称作为结果返回。

① 不过，相关的各种参数都要进行可空性检查，如果你要实现自己的LINQ操作符，牢记这一点十分重要，稍后会在第12章谈到。

说明 **又重复一遍？** 也许你有种似曾相识的感觉。没错，我的确已经在第10章介绍过这些了。但这个话题太重要了，完全有必要再详细地介绍一遍。

这真有点绕嘴，不过序列图可以让它变得更清楚明白。我打算把这些对`MoveNext`和`Current`的调用压缩到一个提取操作中，这可以让图表更加简化。只要记住，每次提取操作执行时，实际上也会检查是否到达序列末尾。图11-2显示了，当我们使用`foreach`循环语句打印结果中的各个元素时，示例查询表达式在运转中的前几个阶段。

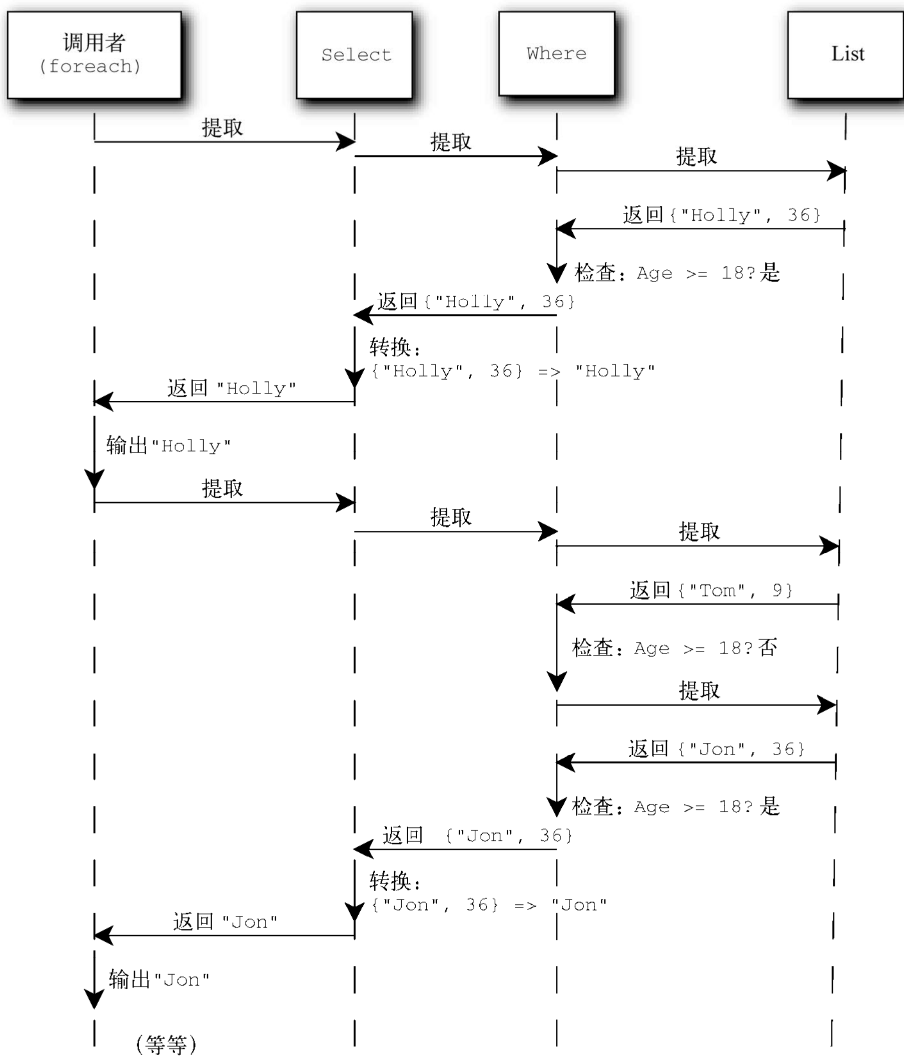


图11-2 查询表达式执行的序列图

正如你在图11-2中看到的，每次只处理数据的一个元素。如果我们决定在输出"Holly"之后停止打印，将不会在原始序列的其他元素上执行任何操作。虽然这里涉及了几个阶段，不过像这样使用流的方式处理数据，是很高效和灵活的。特别是，不管有多少数据源，在某个时间点上，你只需知道一个元素就可以了。

然而，这只是在最好的情况下。有些时候，为了提取查询的第一个结果，你不得不对数据源中所有数据进行计算。在之前的章节中，我们已经看到过这样的一个例子：`Enumerable.Reverse`方法需要提取所有可用的数据，以便把最后一个原始元素作为结果序列的第一个元素返回。这使得`Reverse`成为一个缓冲操作——它在效率上（甚至可行性上）对整个运算都有巨大的影响。如果你无法承受一次性把所有数据都读到内存中的开销，就不能使用缓冲操作。

正如同流处理依赖于你所执行的操作，有些转换一调用就会发生，而不会延迟执行。这称为*立即执行*。一般来说，返回另外一个序列的操作（通常是`IEnumerable<T>`或`IQueryable<T>`）使用延迟执行，而返回单一值的运算使用*立即执行*。

在LINQ中存在着大量的运算，即所谓的*标准查询操作符*——现在让我们来简单地看一下它们。

3. 标准查询操作符

LINQ的*标准查询操作符*是一个转换的集合，具有明确的含义。微软鼓励LINQ提供器尽可能多得实现这些操作符，并让实现符合预期的行为。要跨多个数据源提供一致的查询框架，这是极其重要的。当然，某些LINQ提供器可能暴露了更多功能，而且某些操作符也不一定适当地映射提供器的目标领域——不过至少有机会保持一致。

说明　标准操作符的实现细节　标准查询操作符拥有共同的含义，但这并不代表每个实现工作起来都是完全一样的。例如，有些LINQ提供器可能在查询获取第一个元素时就加载了所有数据——如访问Web服务。同样，LINQ to Objects与LINQ to SQL查询的语义也可能不尽相同。当然，这并不是说LINQ是失败的，只是在我们编写查询时，要考虑访问的是什么样的数据源。拥有一组查询操作符以及一致的查询语法，恰恰是LINQ一个巨大的优势，尽管它不是万能的。

C# 3支持的某些标准查询操作符通过查询表达式内置到语言中，不过你仍然可以手动去调用它们。你或许有兴趣知道，VB 9在语言中具有更多的操作符：不过，在语言中添加新特性所增加的复杂性，与特性带来的好处之间依然需要权衡。我个人认为，C#团队完成了一项值得称赞的工作：我一直喜欢那种背后有庞大函数库来支撑的短小精悍的语言。

说明　操作符“重载”　术语“操作符”可用来描述查询操作符（如`Select`、`Where`方法）和常见的操作符（如相加、相等）。通常从上下文就可以明显地看出我指的是哪个意思——如果正在谈论LINQ，操作符几乎总是指作为查询一部分而使用的方法。

在本章和下一章中，我们会在例子中看到这类操作符，不过我的目标不是做全面的介绍：本书主要是关于C#的，而不是整个LINQ。如要想富有成效地使用LINQ进行产品开发，你并不需要知道所有的操作符，不过你的经验有可能随着时间不断增长。附录A简要描述了各个标准查询操作符，而MSDN为每个特定的重载方法提供了更详细的说明。在你遇到问题的时候，查看一下这个列表：如果觉得应该有一个内置的方法可以帮助你，那么它很可能就真的存在！然而也可能有例外的时候。MoreLINQ开源项目就为LINQ to Objects添加了很多额外的操作符（参见http://code.google.com/p/morelinq/）。同样，Reactive Extensions包（参见http://mng.bz/R7ip）补充了LINQ to Objects的拉模型和推模型，我们将在后面介绍。如果标准操作符不是你想要的，在自己想办法解决前可以先看看这两个项目。如果确实需要自己编写操作符，也没什么可怕的，这一过程充满了乐趣。我们将在第12章讲述这方面的一些技巧。

在接触了一些例子之后，是时候来介绍一下数据模型了，本章剩余的大部分示例代码都将用到它。

11.1.2 定义示例数据模型

在10.3.4节中，我给出了一个缺陷跟踪的简短示例，演示了扩展方法和Lambda表达式的实际用法。本章几乎所有的示例代码也将使用同样的数据模型——它虽然是一个相当简单的模型，但却是一个能以多种不同方式来操纵，从而给出有用的信息的模型。而且，缺陷跟踪也是大部分专业开发人员都十分熟悉的一个业务领域。

我们虚构的场景是SkeetySoft，一个具有远大抱负的小软件公司。创始人决定尝试创建一个办公软件套件、媒体播放器和一个即时消息应用程序。毕竟在这些市场上还没有大牌厂商，有吗？

SkeetySoft的开发部由5个人组成：两个开发人员（Deborah和Darren）、两个测试人员（Tara和Tim）和一个经理（Mary）。目前只有一个客户：Colin。前面提到的产品分别是SkeetyOffice、SkeetyMediaPlayer和SkeetyTalk[①]。数据模型如图11-3所示，我们来查看一下2013年5月里记录的缺陷。

正如你看到的，我们没有记录过多数据。特别是，没有缺陷的真实历史记录，不过这里的内容已经足够我们演示C# 3的查询表达式特性了。 11

基于本章的目的，所有数据都存储在内存中。我们具有一个名为`SampleData`的类，拥有属性`AllDefects`、`AllUsers`、`AllProjects`和`AllSubscriptions`，每个都返回适当的`IEnumerable<T>`类型。`Start`和`End`属性分别返回表示5月份开始和结束时间的`DataTime`实例对象，在`SampleData`中还存在嵌套类`Users`和`Projects`，可以轻松访问特定的用户或项目。有个类型不是那么明显，就是`NotificationSubscription`。它的作用是，每次相关项目的缺陷被创建或改变后，都发一封电子邮件到特定地址。

在示例数据中有41条缺陷，通过C# 3的对象初始化程序来创建。所有的代码都可在本书的网站下载到，不过我不会在本章中包含这些示例数据。

准备工作都已经完成，现在让我们来深入研究一些查询！

---

① SkeetySoft的市场部并不是特别富有创意。

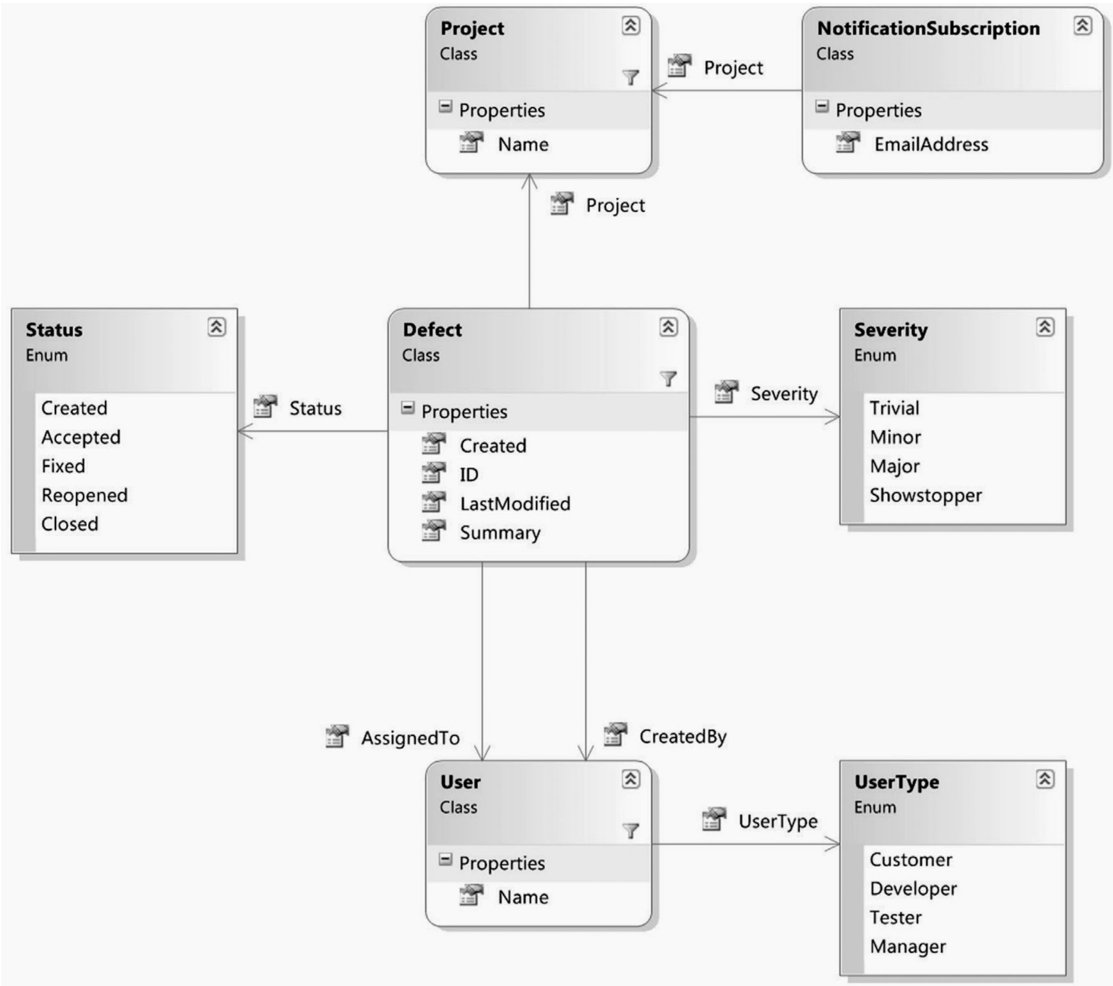


图11-3　SkeetySoft缺陷数据模型的类图

11.2　简单的开始：选择元素

预先学习了一些普通的LINQ概念后，我将介绍一些本章中出现的特定于C# 3的概念。我们从一个简单的查询开始（甚至比之前看到的还要简单），而后逐渐接触一些非常复杂的。这不仅能帮助你理解C# 3编译器所做的工作，还能教会你如何去阅读LINQ代码。

所有的例子都将遵循定义查询，接着把输出结果打印到控制台这样的模式来展开。我们对绑定查询到数据网格或者其他类似的工作不感兴趣——它们虽然重要，却和学习C# 3没有直接关系。

我们通过一个打印出所有用户的简单表达式，来开始研究编译器幕后的执行原理，并学习范围变量。

11.2.1 以数据源作为开始，以选择作为结束

在C# 3中每个查询表达式都以同样的方式开始——声明一个数据序列的数据源：

```
from element in source
```

element只是一个标识符，它前面可以放置一个类型名称。大多数情况下，你都不需要类型名称，因而我们在第一个例子中也省略了。source是一个普通的表达式。在第一个子句出现之后，许多不同的事情会发生，不过迟早都会以一个select子句或group子句来结束。简便起见，我们先使用select子句，它的语法也是很简单的：

```
select expression
```

select子句被称为投影。

它把两者结合在一起，然后用这个微不足道的表达式来作为我们的简单（实际上毫无用处）查询，代码清单11-1将两个子句组合在一起，使用最袖珍的表达式，展示了一个简单而且毫无用处的查询。

代码清单11-1 打印出所有用户的袖珍查询

```
var query = from user in SampleData.AllUsers
            select user;
foreach (var user in query)
{
   Console.WriteLine(user);
}
```

查询表达式就是用粗体标出的那部分。在这个模型中，我们为每个实体重写了ToString方法，所以代码清单11-1就输出如下结果：

```
User: Tim Trotter (Tester)
User: Tara Tutu (Tester)
User: Deborah Denton (Developer)
User: Darren Dahlia (Developer)
User: Mary Malcop (Manager)
User: Colin Carton (Customer)
```

你可能会想，这样一个例子有什么用？毕竟，我们是在foreach语句中直接使用SampleData.AllUsers。然而，我们可以用这个查询表达式（虽然有点袖珍）来介绍两个新概念。首先，我们会看到编译器在遇到查询表达式的时候，使用转换过程的常见特点；接着，我们会讨论范围变量。

11.2.2 编译器转译是查询表达式基础的转译

编译器把查询表达式转译为普通的C#代码，这是支持C# 3查询表达式的基础。它是以一种机

械的方式来进行转换的，不会去理解代码、应用类型引用、检查方法调用的有效性或执行编译器要执行的任何正常工作。这些都在转换完成之后来执行。从许多方面看，第一个阶段可以看做是预处理器阶段。

编译器在执行真正的编译之前把代码清单11-1转译为代码清单11-2。

代码清单11-2　代码清单11-1的查询表达式被转译为一个方法调用

```
var query = SampleData.AllUsers.Select(user => user);

foreach (var user in query)
{
   Console.WriteLine(user);
}
```

C# 3编译器进一步正确地编译查询表达式之前，就把它转译为了这样的代码。特别地，它不会假定应该使用`Enumerable.Select`，还是`List<T>`应该包含一个`Select`方法调用。它只是对代码进行转译,并让下一个编译阶段来处理查找适当方法的工作——不管这个方法是直接包含的成员，还是扩展方法[①]。参数是一个适当的委托类型或者一个和类型`T`对应的`Expression<T>`。

重要之处在于，Lambda表达式能被转换为委托实例和表达式树。本章中所有的例子都将使用委托，不过在第12章中学习其他LINQ提供器的时候，我们将会看到表达式树是如何使用的。稍后，在我介绍某些由编译器调用的方法的签名时，记住在LINQ to Objects中只进行一种调用——任何时候，参数（大部分）都是委托类型，编译器将用Lambda表达式作为实参，并尽量查找具有合适签名的方法。

还必须记住，在转译执行之后，不管Lambda表达式中的普通变量（比如方法内部的局部变量）在哪出现，它都会以我们在第5章看到的方式转换成捕获变量。这只是普通Lambda表达式的行为——不过除非你理解哪些变量将被捕获，否则很容易被查询结果弄糊涂。

语言规范给出了查询表达式模式的细节，必须实现所有查询表达式的查询表达式模式，才能正常工作，不过它并没有如你所期望的那样被定义为一个接口。然而，这样是很有道理的：它通过扩展方法，能够让LINQ应用于`IEnumerable<T>`这样的接口。本章将逐一讲解查询表达式模式的每个元素。如果你想看看语言规范是如何定义各个转译的，可以参考C# 4规范的7.16节。

代码清单11-3证实了编译器转译的工作原理：它为`Select`和`Where`提供了伪实现，使`Select`成为一个普通的实例方法，而使`Where`成为一个扩展方法。我们最初的简单查询表达式只包含了`Select`子句，不过现在我们要来包含`Where`子句来展示两种方法的使用。可由于扩展方法需要声明在静态类中，所以我给出了完整的代码清单，而不再是代码片段。

① 实际的情况比这要更加宽泛——编译器不要求`Select`必须为一个方法，或`SampleData.AllUsers`必须为属性。只要转换后的代码可以编译就可以了。在大多数情况下，你都会访问标准方法或扩展方法，但我写了一篇博客，介绍了几个奇特的查询（参见http://mng.bz/7E3i）。在实际项目中，我还没发现这样的查询有多大用处，但它可以说明转换过程的机制，以及为什么无须关注转换后代码的含义。

代码清单11-3 编译器转译调用伪LINQ实现中的方法

```
static class Extensions
{
   public static Dummy<T> Where<T>(this Dummy<T> dummy,          <-- ❶ 声明Where扩展方法
                                   Func<T,bool> predicate)
   {
      Console.WriteLine("Where called");
      return dummy;
   }
}

class Dummy<T>
{
   public Dummy<U> Select<U>(Func<T,U> selector)          <-- ❷ 声明Select实例方法
   {
      Console.WriteLine("Select called");
      return new Dummy<U>();
   }
}

class TranslationExample
{
   static void Main()
   {
      var source = new Dummy<string>();          <-- ❸ 创建用于查询的数据源
      var query = from dummy in source          <-- ❹ 通过查询表达式来调用方法
                  where dummy.ToString() == "Ignored"
                  select "Anything";
   }
}
```

运行代码清单11-3首先会打印出`Where called`，接着会打印`Select called`，一切正如我们所预期那样，由于查询表达式已经被转译为如下的这段代码：

```
var query = source.Where(dummy => dummy.ToString() == "Ignored")
                  .Select(dummy => "Anything");
```

当然，我们在这里没有完成任何查询或转换，不过还是展示了编译器对我们的查询表达式进行的转译。假如你对为何要选择`"Anything"`而不是dummy感到迷惑的话，我可告诉你，这是因为仅仅用dummy作为投影的这种特殊情况（这是一个“无所作为”的投影）会被编译器删除。我们将在后面的11.3.2节看到这种情况，不过现在的重要目的是转换涉及的所有类型。我们只需要学习C#编译器将使用什么转译，这样我们就能书写任意查询表达式，并把它转换为不使用查询表达式的形式，并从这个角度来研究一下它都做了什么。

注意，在`Dummy<T>`中根本没有实现`IEnumerable<T>`。从查询表达式到“普通”代码的转换并不依赖于它，而实际上几乎所有LINQ提供器都把数据显示为`IEnumerable<T>`或`IQueryable<T>`（将在第12章中看到）。转译不依赖于任何特定类型而仅仅依赖于方法名称和参数，这是一种鸭子类型（Duck Typing）的编译时形式。这和第8章中介绍的集合初始化程序使用了同样的方式：使用普通的重载决策来查找公共方法调用`Add`，而不是使有包含特定签名的`Add`方法的接口。查询表达式进一步利用了这种思想——转译发生在编译过程初期，以便让编译器来

挑选实例方法或者扩展方法。你甚至可以认为，转译是在一个独立的预处理引擎中工作。

你可能觉得我在此过于啰嗦，但这些才是让LINQ拨云见日的部分。如果你将查询表达式重写为一系列的方法调用，编译器所做的事情没有变化，性能也不会改变，查询的行为也不会有什么异常。它们只是表示相同代码的两种不同的方式。

说明　为什么写成from...where...select，而不是写成select...from...where？ 很多开发人员一开始都困惑于查询表达式中子句的顺序。它就像SQL语句——只是前后颠倒。如果你回顾一下转译到方法的过程，你就会明白背后的主要原因。查询表达式以书写的顺序来被处理：我们从from子句中的数据源开始，在where子句中进行过滤，然后在select子句中进行投影。另外一种理解方法就是，思考一下贯穿本章的那个图表。在从上到下的数据流中，图表中的框的出现顺序和在查询表达式中对应的子句的出现顺序是相同的。一旦你克服了最初的由陌生引起的不适，你就会发现这是一种很好的方法——我就是这样。你甚至可能会问为什么SQL不是这样的顺序。

到此，我们已经明白了，这里进行了数据源级别的转译。不过在进一步学习之前，还有另外一个重要的概念需要理解。

11.2.3　范围变量和重要的投影

让我们更深入地回想一下最初的查询表达式。我们并没有检查from子句中的标识符或select子句中的表达式。图11-4再次显示了这个查询表达式，每个部分都用使用标注来解释其目的。

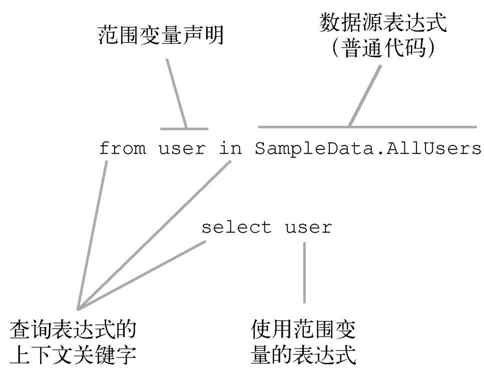


图11-4　一个被分解成了多个组成部分的简单查询表达式

上下文关键字很容易解释——它们明确告知编译器我们要对数据进行的处理。同样，数据源表达式也仅仅是普通的C#表达式——在这个例子中是一个属性，不过它也可以是一个简单的方法调用或变量。

这里较难理解的是范围变量声明和投影表达式。范围变量不像其他种类的变量。在某些方面，它根本就不是变量。它们只能用于查询表达式中，实际代表了从一个表达式传递给另外一个表达式的上下文信息。它们表示了特定序列中的一个元素，而且它们被用于编译器转译中，以便把其他表达式轻易地转译为Lambda表达式。

我们已经知道最初的查询表达式会转换为如下形式：

```
SampleData.AllUsers.Select(user => user)
```

Lambda表达式的左边——提供参数名称的部分来自于范围变量的声明。而右边来自于select子句。这个转译过程就是这么简单（本例是这样）。这样一切都没有问题，因为我们在两边都使用了同样的名称。假如我们按如下方式书写查询表达式：

```
from user in SampleData.AllUsers
select person
```

在这种情况下，转换后的版本应该是：

```
SampleData.AllUsers.Select(user => person)
```

在这里，编译器会出错，因为它不知道person指的是什么。到此，我们知道转换过程是多么简单，然后这样也可让那些具有更复杂投影的查询表达式变得易于理解。代码清单11-4打印出user对象的名字。

代码清单11-4　仅选择user对象名称的查询

```
IEnumerable<string> query = from user in SampleData.AllUsers
                            select user.Name;

foreach (string name in query)
{
   Console.WriteLine(name);
}
```

这次，我们使用user.Name作为投影，即可看到结果变为字符串序列，而不是User对象了。（我曾用显式类型变量来强调这一点）查询表达式的转换遵循与之前一样的规则，则变为：


```
SampleData.AllUsers.Select(user => user.Name)
```

编译器准许这样做，是由于从Enumerable所选择的Select扩展方法实际上具有如下签名[①]：

```
static IEnumerable<TResult> Select<TSource,TResult>
   (this IEnumerable<TSource> source,
    Func<TSource,TResult> selector)
```

在将Lambda表达式转换为Func<TSource, TResult>的时候，第9章介绍的类型推断也发挥了作用。它首先根据SampleData.AllUsers的类型推断出TSource为User，这样就知道了Lambda表达式的参数类型，并因此将user.Name作为返回string类型的属性访问表达式，也就可以推断出TResult为string。这就是Lambda表达式允许使用隐式类型参数的原因，也就是会存在如此复杂的类型推断规则的原因：这些都是LINQ引擎的“齿轮”和“活塞”。

① 篇幅所限，我忽略了本章所有方法前面的public修饰符。实际上，它们都是公共的。

说明　为什么你需要知道这些东西？　多数时候，你都几乎可以忽略范围变量发生的事情。你很可能见到过很多很多的查询，都明白它们要完成的事情，而无须知道背后发生的一切。在查询正常工作的时候（正如教程中的那些例子一样），这毫无问题，不过当它发生错误的时候，就值得去了解一些细节的东西。如果你有一个不能编译的查询表达式，因为编译器报错说它不知道某个特定的标识符，那么你应该查看一下其中的范围变量。

到目前为止，我们只是研究了一些隐式类型的范围变量。当我们在声明中包含类型的时候，会发生些什么？答案就在Cast和OfType标准查询操作符中。

11.2.4　Cast、OfType和显式类型的范围变量

大多数时候，范围变量都可以是隐式类型。但你可能使用所需类型全部指定了的泛型集合。可是，万一不是这种情况呢？如果我们想对一个ArrayList或一个object[]执行查询呢？假如LINQ不能处理这种情况，那么就有点可惜了。幸好，有两个标准查询操作符来解决这个问题：Cast和OfType。只有Cast是查询表达式语法直接支持的，不过我们在本节中会对两者进行研究。

这两个操作符很相似：都可以处理任意非类型化的序列（它们是非泛型IEnumerable类的扩展方法），并返回强类型的序列。Cast通过把每个元素都转换为目标类型（遇到不是正确类型的任何元素的时候，就会出错）来处理，而OfType首先进行一个测试，以跳过任何具有错误类型的元素。

代码清单11-5演示了这两个操作符，作为Enumerable中的一个简单的扩展方法来使用。为了有所改变，我们不使用SkeetySoft缺陷系统作为我们的示例数据——毕竟，它们全部都是强类型！相反，我们仅仅使用两个ArrayList对象。

代码清单11-5　使用Cast和OfType来处理弱类型集合

```
ArrayList list = new ArrayList { "First", "Second", "Third" };
IEnumerable<string> strings = list.Cast<string>();
foreach (string item in strings)
{
   Console.WriteLine(item);
}

list = new ArrayList { 1, "not an int", 2, 3 };
IEnumerable<int> ints = list.OfType<int>();
foreach (int item in ints)
{
   Console.WriteLine(item);
}
```

第1个列表只包括字符串，所以可以放心地使用Cast<string>来获得一个字符串序列。第2个列表包含混杂的内容，所以为了从中只获取整数，我们只能使用OfType<int>。如果我们在第2个列表上面使用Cast<int>，那么在尝试把“not an int”转换为int的时候，就会抛出一个异常。注意，这个异常只会发生在打印出“1”之后——两个操作符都对数据进行流处理，在获取

元素的时候才对其进行转换。

> **说明** **只允许一致性、引用和拆箱转换** .NET 3.5 SP1修改了Cast的行为。在原来的.NET 3.5中，允许执行的转换很多，你可以对List<short>使用Cast<int>将每个short转换成int。而在SP1中，这样将抛出异常。如果需要引用转换和拆箱转换（或非操作的一致性转换）之外的转换，可以使用Select投影。OfType也只能执行这些转换，不过失败时它不会抛出异常。

在你引入了具有显式类型的范围变量后，编译器就调用Cast来保证在查询表达式的剩余部分中使用的序列具有合适的类型。代码清单11-6显示了这个过程，并通过使用Substring方法进行投影，来证明由from子句生成的序列是一个字符串序列。

代码清单11-6 使用显式类型的范围变量来自动调用Cast

```
ArrayList list = new ArrayList { "First", "Second", "Third"};
var strings = from string entry in list
              select entry.Substring(0, 3);
foreach (string start in strings)
{
   Console.WriteLine(start);
}
```

代码清单11-6的输出结果是Fir、Sec、Thi，不过更有意思的是转译过的查询表达式，如下所示：

```
list.Cast<string>().Select(entry => entry.Substring(0,3));
```

没有这个类型转换，我们根本就不能调用Select，因为该扩展方法只用于IEnumerable<T>而不能用于IEnumerable。即使使用了强类型集合，你可能还是想使用显式类型的范围变量。例如，你有一个集合定义为List<ISomeInterface>，不过你知道其中所有的元素都是MyImplementation的实例。那么，使用显式类型为MyImplementation的范围变量，就可以访问MyImplementation的所有成员，而不用在所有地方手动插入类型转换代码。

目前，尽管我们尚未查询出任何给人留下深刻印象的结果，但还是涉及了很多重要概念。我们再次简要地概括一下这些重要概念。

- LINQ以数据序列为基础，在任何可能的地方都进行流处理。
- 创建一个查询并不会立即执行它：大部分操作都会延迟执行。
- C# 3的查询表达式包括一个把表达式转换为普通C#代码的预处理阶段，接着使用类型推断、重载、Lambda表达式等这些常规的规则来恰当地对转换后的代码进行编译。
- 在查询表达式中声明的变量的作用：它们仅仅是范围变量，通过它们你可以在查询表达式内部一致地引用数据。

我知道，这里面包括许多稍稍有点抽象的信息。不用担心和怀疑，LINQ值得面对这些小问题。我可以向大家做出承诺。在学习了这么多基础知识以后，我们就可以开始来真正地做一些有

11

用的事情了——比如先过滤数据，接着进行排序。

11.3　对序列进行过滤和排序

学习这两个操作时，你可能会感到惊讶，因为它们是对编译器转换进行解释的最简单的方法之一。原因在于，它们总是返回和输入同样类型的序列，这意味着我们不需要担心会引入新的范围变量。我们在第10章提到的相应扩展方法对此也会有所帮助。

11.3.1　使用where子句进行过滤

要弄懂where子句非常容易。它的语法格式如下：

```
where 过滤表达式
```

编译器把这个子句转译为带有Lambda表达式的Where方法调用，它使用合适的范围变量作为这个Lambda表达式的参数，而以过滤表达式作为主体。过滤表达式当作进入数据流的每个元素的谓词，只有返回true的元素才能出现在结果序列中。使用多个where子句，会导致多个链接在一起的Where调用——只有满足所有谓词的元素才能进入结果序列。代码清单11-7演示了查找分配给Tim的所有开口缺陷的查询表达式。

代码清单11-7　使用多个where子句的查询表达式

```
User tim = SampleData.Users.TesterTim;

var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            select defect.Summary;
foreach (var summary in query)
{
   Console.WriteLine(summary);
}
```

代码清单11-7中的查询表达式被转译为：

```
SampleData.AllDefects.Where (defect => defect.Status != Status.Closed)
                     .Where(defect => defect.AssignedTo == tim)
                     .Select(defect => defect.Summary)
```

代码清单11-7输出的结果如下：

```
Installation is slow
Subtitles only work in Welsh
Play button points the wrong way
Webcam makes me look bald
Network is saturated when playing WAV file
```

当然，我们可以编写合并两个条件的单一where子句，来替换多个where子句的使用。在某些情况下，这样可以提高性能，但也要考虑查询表达式的可读性。再次声明，这很可能是主观的看法。我个人更倾向于把逻辑上关联的条件合并在一起，而逻辑上不相关的保持独立。在这个例子中，表达式的两个部分都是直接处理缺陷（都是作为序列的限制条件），所以把它们合并到一

起更合理。和以前一样，使用两种形式只是为了让大家明白哪一种形式使结构更清晰。

我们马上就要开始来尝试把排序规则应用到我们的查询上，不过首先，应该研究一下使用select子句过程中的一些小细节。

11.3.2 退化的查询表达式

虽然我们可以使用一种相当简单的转译，但还是要重新回顾一下，在11.2.2节当我第一次介绍编译器转换的时候，没有提到的一些细节问题。到目前为止，我们所有转换后的查询表达式都包含了对Select的调用。如果我们的select子句什么都不做，只是返回同给定的序列相同的序列，会发生什么？答案是编译器会删除所有对Select的调用。当然，前提是在查询表达式中还有其他操作可执行时才这么做。

例如，下面的查询表达式选择了在系统中的所有缺陷：

```
from defect in SampleData.AllDefects
select defect
```

这就是所谓的*退化查询表达式*。编译器会故意生成一个对Select方法的调用，即使它什么都没有做：

```
SampleData.AllDefects.Select(defect => defect)
```

然而，这个代码和以SampleData.AllDefects作为一个简单表达式还是有很大不同。两个序列返回的数据项是相同的，但是Select方法的结果*只*是数据项的序列，而不是数据源本身。查询表达式的结果和源数据永远不会是同一个对象，除非LINQ提供器的代码有问题。从数据集成的角度看，这是很重要的——提供器能返回一个易变的结果对象，并知道即使面对一个退化查询，对返回数据集的改变也不会影响到“主”数据。

当有其他操作存在的时候，就不用为编译器保留一个“空操作”的select子句了。例如，假设我们把代码清单11-7中的查询表达式改为选取整个缺陷而不仅仅是名称：

```
from defect in SampleData.AllDefects
where defect.Status != Status.Closed
where defect.AssignedTo == SampleData.Users.TesterTim
select defect
```

现在我们不需要最后的Select调用，所以转换后的代码如下：

```
SampleData.AllDefects.Where(defect => defect.Status != Status.Closed)
                     .Where(defect => defect.AssignedTo == tim)
```

在你编写查询表达式的时候，这些规则很少会妨碍你，不过如果你使用诸如Reflector这样的工具来反编译代码的话，你可能会对此感到迷惑——Select的调用无缘无故地不见了，这确实会让人感到惊讶。

掌握这个知识点后，现在来改进一下我们的查询，以便知道Tim下一步应该干什么。

11.3.3 使用**orderby**子句进行排序

对于开发人员和测试人员来说，他们经常被要求在处理一些不重要的缺陷之前，先着手去解

决最重要的缺陷。我们可以使用一个简单查询来告知Tim，他应该处理分配给他的开口缺陷的顺序。代码清单11-8使用orderby子句以优先级降序的方式，完全按照我们所设想的那样，打印出了所有缺陷的详细信息。

代码清单11-8　按缺陷严重度的优先级从高到低的顺序排序

```
User tim = SampleData.Users.TesterTim;

var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            orderby defect.Severity descending
            select defect;

foreach (var defect in query)
{
   Console.WriteLine("{0}: {1}", defect.Severity, defect.Summary);
}
```

代码清单11-8的输出结果显示，我们对结果进行了适当地排序。

```
Showstopper: Webcam makes me look bald
Major: Subtitles only work in Welsh
Major: Play button points the wrong way
Minor: Network is saturated when playing WAV file
Trivial: Installation is slow
```

我们看到已经获得两个主要的缺陷，它们的顺序应该是什么样的呢？目前还没有进行明确的排序。

让我们改变一下查询，以便在按照严重性降序排序后，能按照“最后修改时间”来进行升序排序。这意味着，Tim将先测试那些之前已经修正了很久的缺陷，然后再测试最近修正的缺陷。这只需在orderby子句中要加入额外的表达式，如代码清单11-9所示。

代码清单11-9　先按严重度排序，而后按最后修改时间排序

```
User tim = SampleData.Users.TesterTim;

var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            orderby defect.Severity descending, defect.LastModified
            select defect;
foreach (var defect in query)
{
   Console.WriteLine("{0}: {1} ({2:d})",
                defect.Severity, defect.Summary, defect.LastModified);
}
```

代码清单11-9的运行结果显示在下面。注意，两个主要缺陷的顺序是如何被颠倒的。

```
Showstopper: Webcam makes me look bald (05/27/2013)
Major: Play button points the wrong way (05/17/2013)
Major: Subtitles only work in Welsh (05/23/2013)
Minor: Network is saturated when playing WAV file (05/31/2013)
Trivial: Installation is slow (05/15/2013)
```

所以，查询表达式就变成了现在这样——不过，编译器都做了些什么呢？它只是简单调用了OrderBy和ThenBy方法（或用于降序的OrderByDescending/ThenByDescending）。我们的查询表达式转译成了：

```
SampleData.AllDefects.Where(defect => defect.Status != Status.Closed)
                     .Where(defect => defect.AssignedTo == tim)
                     .OrderByDescending(defect => defect.Severity)
                     .ThenBy(defect => defect.LastModified)
```

在看过这个例子之后，让我们来研究一下orderby子句的一般语法。它们基本上是上下文关键字orderby，后面跟一个或多个排序规则。一个排序规则就是一个表达式（可以使用范围变量），后面可以紧跟ascending或descending关键字，它的意思显而易见（默认规则是升序。）对于主排序规则的转译就是调用OrderBy或OrderByDescending，而其他子排序规则通过调用ThenBy或ThenByDescending来进行转换，正如我们例子中看到的。

OrderBy和ThenBy的不同之处非常简单：OrderBy假设它对排序规则起决定作用，而ThenBy可理解为对之前的一个或多个排序规则起辅助作用。对于LINQ to Objects而言，ThenBy只是定义为IOrderedEnumerable<T>的扩展方法，这是一个由OrderBy返回的类型（ThenBy本身也会返回这个类型，以便支持进一步的链接）。

要特别注意，尽管你能使用多个orderby子句，但每个都会以它自己的OrderBy或OrderByDescending子句作为开始，这意味着最后一个才会真正“获胜”。也许会存在某些原因要包括多个orderby子句，不过这是很少见的。一般都应该使用单个子句来包含多个排序规则。

在第10章提到过，应用排序规则要求所有数据都已经载入（至少对于LINQ to Objecs是这样的）——例如，你就不能对一个无限序列进行排序。这个原因是显而易见的，比如，在你看到所有元素之前，你不知道你看到的某些东西是否出现在结果序列的开头。

我们大约学习了查询表达式一半的内容，你也许会惊讶，因为我们还没有看到任何连接（join）操作。显然，它们在LINQ中的重要性和在SQL中一样，不过它们也是很复杂的。我保证在时机成熟的时候一定会谈到它们，不过为了每次只介绍一个新概念，我们需要先来学习一下let子句。在接触连接操作之前，我们还会学习透明标识符。

11

11.4 **let** 子句和透明标识符

接下来我们将要研究的大部分操作符都会涉及*透明标识符*。就像范围变量那样，如果你只想对查询表达式有个表层的了解，那么无须深刻理解透明标识符，就可毫无障碍地使用它。但既然你购买了本书，我希望你深入地了解C#，这样你就能正视编译错误，并明白它们指的是什么。

你不用知道透明标识符的*所有方面*，不过我将教你足够的知识，以便你在语言规范中看到某个透明标识符的时候，不会想着躲避。你也将会了解到，它们为何是必需的——我们将用一个例子来解释。let子句是最简单的使用透明标识符的转换。

11.4.1 用**let**来进行中间计算

let子句只不过引入了一个新的范围变量，它的值是基于其他范围变量。语法是极其简单的：

```
let 标识符=表达式
```

要明确地解释let，而不使用任何其他复杂的操作符，我打算用一个非常不实际的例子来阐明。假设查找字符串长度是一种非常昂贵的操作。现在，想象我们有一个非常奇异的系统需求：按照用户名称的长度来排序，并显示名称及长度。是的，我知道这是有点不太可能。代码清单11-10显示了不用let子句的情况下的实现方法。

代码清单11-10　在不使用let子句的情况下，按用户名称的长度来排序

```
var query = from user in SampleData.AllUsers
            orderby user.Name.Length
            select user.Name;

foreach (var name in query)
{
   Console.WriteLine("{0}: {1}", name.Length, name);
}
```

这段代码运行正常，不过它调用了“可怕的”Length属性两次——一次是对用户进行排序，一次用于显示。毫无疑问，就算是最快的超级计算机也无法处理两次查找6个字符的字符串长度的操作！不能这样，我们需要避免这个冗余的计算。

这正是let子句的用武之地，它对一个表达式进行求值，并引入一个新的范围变量。代码清单11-11与代码清单11-10的结果相同，不过对于每个用户只使用了一次Length属性。

代码清单11-11　使用let子句来消除冗余的计算

```
var query = from user in SampleData.AllUsers
            let length = user.Name.Length
            orderby length
            select new { Name = user.Name, Length = length };

foreach (var entry in query)
{
   Console.WriteLine("{0}: {1}", entry.Length, entry.Name);
}
```

代码清单11-11引入了一个名为length的范围变量，它包含了用户名的长度（针对原始序列中的当前用户）。我们接着把新的范围变量用于排序和最后的投影。你发现问题了吗？我们需要使用两个范围变量，但Lambda表达式只会给Select传递一个参数！这就该透明标识符出场了。

11.4.2　透明标识符

在代码清单11-11中，我们在最后的投影中使用了两个范围变量，不过Select方法只对单个序列起作用。如何把范围变量合并在一起呢？

答案是，创建一个匿名类型来包含两个变量，不过需要进行一个巧妙的转换，以便看起来就像在select和orderby子句中实际应用了两个参数。图11-5显示了其中的过程。

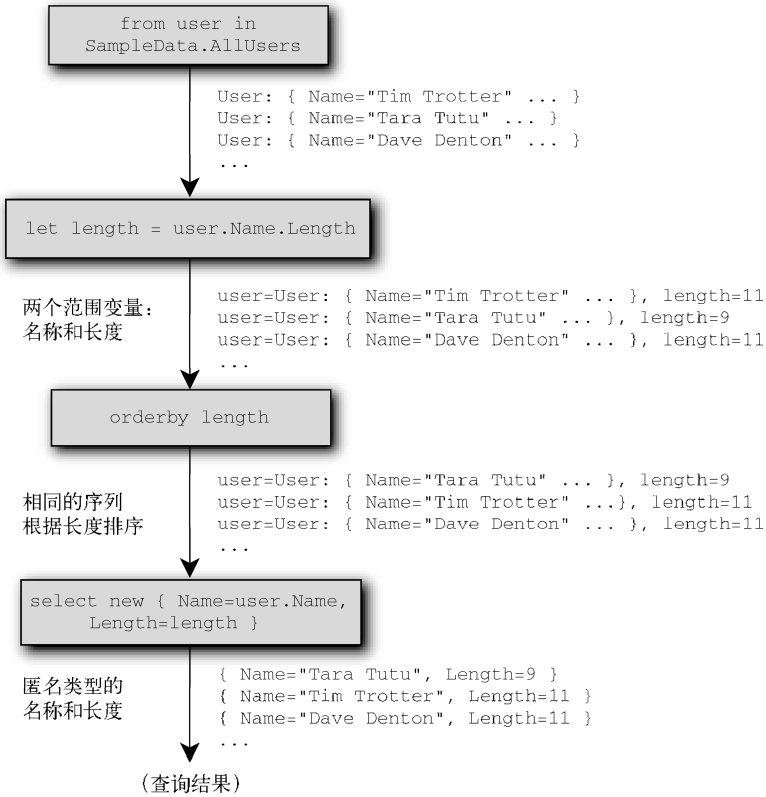


图11-5　代码清单11-11的执行过程，其中let子句引入了length范围变量

let子句为了实现目标，再一次调用了Select，并为结果序列创建匿名类型，最终创建了一个新的范围变量（它的名称在源代码中从未看到或用到）。代码清单11-11中的查询表达式被转换为如下内容：

```
SampleData.AllUsers
          .Select(user => new { user, length = user.Name.Length })
          .OrderBy(z => z.length)
          .Select(z => new { Name = z.user.Name, Length = z.length })
```

查询的每个部分都进行了适当的调整：对于原始的查询表达式直接引用user或length的地方，如果引用发生在let子句之后，就用z.user或z.length来代替。这里z这个名称是随机选择的——一切都被编译器隐藏起来。

说明 **匿名类型只是一种实现方式** 严格地说，如何将不同的范围变量组合到一起，并使透明标识符得以工作，取决于C#编译器的实现。微软的官方实现使用了匿名类型，规范中也展示了这种转换——所以我也紧跟潮流。即使其他的编译器使用了不同的方式，也不应该影响结果。

如果阅读C#语言规范关于let子句的内容（C# 4规范7.16.2.4节），你将看到它描述的转译是从一个查询表达式到另外一个查询表达式。它使用了星号（*）来表示引入的透明标识符。透明标识符在转译的最后一步被清除。我在本章没有使用该符号，因为它很难使用，而且对于我们打算深入到的细节水平来说，它也是不必要的。希望在了解这一背景知识后，这个规范不再显得那么高深莫测，也可以让你在需要的时候参考一下。

现在我们终于可以学习C# 3查询表达式支持的余下的转译，这确实是个好消息。我不想详细讲解引入的每个透明标识符，不过在遇到的时候，我会简单提及。首先，让我们来看一下对连接的支持。

11.5　连接

如果你学习过SQL，应该对数据库连接有个大致的概念。它使用两张数据表（或视图、表值函数等），通过匹配两者之间的数据行来创建结果。LINQ的连接与之类似，只不过操作的是序列。有3种类型的连接，然而它们并不是都在查询表达式中使用join关键字。首先来看一种最接近SQL内连接的连接。

11.5.1　使用join子句的内连接

内连接涉及两个序列。一个键选择器（key selector）表达式应用于第1个序列的每个元素，另外一个键选择器（可能完全不同）应用于第2个序列的每个元素。连接的结果是一个包含所有配对元素的序列，配对的规则是第1个元素的键与第2个元素的键相同。

说明　术语冲突：内联和外联序列　MSDN文档在描述用于计算内连接的Join方法时，将相关的序列称为inner和outer。实际的方法参数也基于这两个名称。这与内连接和外连接没有多大关系——它只是区分序列的一种方法。你可以把它们看做第1个和第2个，左边和右边，Bert和Ernie——任何你喜欢的能够帮助你区分的东西。本章，我将使用左和右进行区分，这样就可以用图的形式来明确不同的序列。通常来说，outer与左连接对应，inner与右连接对应。

这两个序列可以是任何你喜欢的东西：如果有必要的话，右边的序列甚至可以和左边的一样（例如，要找出同一天出生的人）。唯一要注意的问题是，两个键选择器表达式必须有相同的键类型[①]。

你不能说让人的出生日期和城市的人口数相等，而把人的序列和城市的序列连接一起——这

① 如果两个键类型可以隐式地从其中一个转换到另外一个，也是有效的。其中的一个类型相比另一个类型来说，必须是更好的选择。编译器在推断隐式类型数组的类型时，也是使用同样的方式。以我的经验来说，你很少需要有意识地考虑这个细节问题。

样毫无意义。我们也完全有可能用匿名类型来作为键，因为匿名类型实现了适当的相等性和散列。如果想创建一个多列的键，就可以使用匿名类型。它也可以用于分组操作，稍后将会介绍。

内连接的语法看上去似乎很复杂：

```
[查询选择左边的序列]
join right-range-variable in right-sequence
   on left-key-selector equals right-key-selector
```

在看到使用`equals`而不是符号作为上下文关键字的时候，会感到有点不适应，不过这样也更容易把左键选择器和右键选择器区分开。通常（但不总是）至少有一个键选择器是不那么重要的，因为它只是从序列中选取确切的元素。编译器使用这个上下文关键字将键选择器划分为不同的Lambda表达式。为每个值获取各个键的值（左连接或右连接）的功能是十分重要的，不管是对于LINQ to Objects的性能，还是将查询转译为其他形式（如SQL）的可行性来说，都是如此。

还是来看一个关于缺陷系统的例子。假如我们已经添加了通知功能，并希望发出针对所有现存缺陷的第一批邮件。我们需要把项目相同的通知列表和缺陷列表连接在一起。代码清单11-12就完成了这样一个连接操作。

代码清单11-12　根据项目把缺陷和通知订阅连接在一起

```
var query = from defect in SampleData.AllDefects
            join subscription in SampleData.AllSubscriptions
               on defect.Project equals subscription.Project
            select new { defect.Summary, subscription.EmailAddress };

foreach (var entry in query)
{
   Console.WriteLine("{0}: {1}", entry.EmailAddress, entry.Summary);
}
```

代码清单11-12将显示每个媒体播放器缺陷两次——一次显示“mediabugs@skeetysoft.com”，一次显示theboss@skeetysoft.com（因为老板非常关心媒体播放器项目）。

在这个特别的例子中，我们很容易进行反转连接，调换左右序列的位置。结果包含的条目相同，只是顺序不同。在LINQ to Objects的实现中，返回条目的顺序为：先返回使用左边序列中第1个元素的所有成对数据能被返回（按右边序列的顺序），接着返回使用左边序列中第2个元素的所有成对数据，依次类推。右边序列被缓冲处理，不过左边序列仍然进行流处理——所以，如果你打算把一个巨大的序列连接到一个极小的序列上，应尽可能把小序列作为右边序列。这种操作仍然是延迟的：在访问第1个数据对时，它才会开始执行，然后再从某个序列中读取数据。这时，它会读取整个右边序列，来建立一个从键到生成这些键的值的映射。之后，它就不需要再次读取右边的序列了，这时你可以迭代左边的序列，生成适当的数据对。

可能出错的地方是，把键选择器放到了相反的位置。在左边的键选择器中，只能访问左边序列的范围变量；在右边的键选择器中，只能访问右边序列的范围变量。如果你颠倒了左右两边的序列，那么你必须也颠倒左右两边的键选择器。幸好，编译器知道这样的常见错误，并建议你按恰当的步骤进行处理。

为了使所发生的事情更加明显，图11-6显示了序列的处理过程。

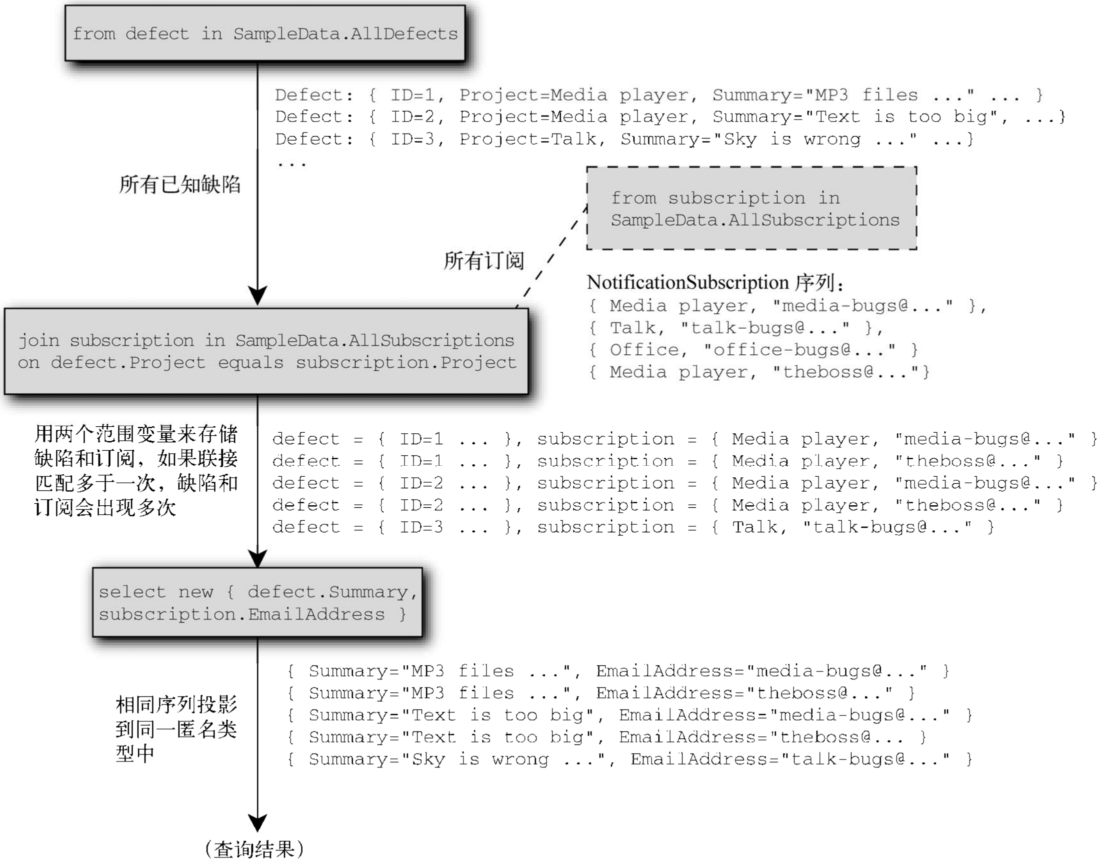


图11-6　代码清单11-12中的连接，展现了用作数据源的两个不同序列（缺陷和订阅）

我们通常需要对序列进行过滤，而在连接前进行过滤比在连接后过滤效率要高得多。在这个阶段，假如只有左边的序列需要进行过滤，查询表达式相对简单一些。例如，如果我们只想显示已经关闭的缺陷，我们可以使用下面的查询表达式：

```
from defect in SampleData.AllDefects
where defect.Status == Status.Closed
join subscription in SampleData.AllSubscriptions
   on defect.Project equals subscription.Project
select new { defect.Summary, subscription.EmailAddress }
```

我们也能在反向的序列上执行同样的查询，不过稍显麻烦：

```
from subscription in SampleData.AllSubscriptions
join defect in (from defect in SampleData.AllDefects
                where defect.Status == Status.Closed
                select defect)
   on subscription.Project equals defect.Project
select new { defect.Summary, subscription.EmailAddress }
```

注意，如何在查询表达式中使用另一个查询表达式——实际上，语言规范描述了许多这种形式的编译器转译。嵌套查询表达式非常有用，不过也会降低可读性：通常应该寻找一种替代方式，或者将右边序列赋给一个变量，来让代码更加清晰。

说明　内连接在LINQ to Objects中很有用吗？　SQL总是会使用内连接。它们实际上是从某个实体导航到相关联的实体上的一种方式，通常是把某个表的外键和另外一个表的主键进行连接。在面向对象模型中，我们倾向于通过引用来从某个对象导航到另外一个对象。例如，在SQL中，要得到缺陷的概要和处理这个缺陷的用户名称，需要进行连接——在C#中，我们则使用属性链。如果在我们的模型中存在一个反向关联，从Project对象到与之关联的NotificationSubscription对象列表，我们不必使用连接也可以实现这个例子的目标。这并不是说，在面向对象模型里面，内联没有用——只是没有在关系模型中出现得那么频繁而已。

内联被编译器转译为对Join方法的调用，如下所示：

```
leftSequence.Join(rightSequence,
                  leftKeySelector,
                  rightKeySelector,
                  resultSelector)
```

用于LINQ to Objects的重载签名如下（这是真正的方法签名，包含真正的参数名称，因此使用了内引用和外引用）：

```
static IEnumerable<TResult> Join<TOuter,TInner,TKey,TResult> (
   this IEnumerable<TOuter> outer,
   IEnumerable<TInner> inner,
   Func<TOuter,TKey> outerKeySelector,
   Func<TInner,TKey> innerKeySelector,
   Func<TOuter,TInner,TResult> resultSelector
)
```

11

当你记得把内连接和外连接分别看做是*右边*和*左边*时，前3个参数的含义显而易见，不过最后一个参数更加有趣。这是从两个元素（一个来自于左边序列，一个来自于右边序列）到结果序列中单个元素的投影。

当连接的后面不是select子句时，C# 3编译器就会引入透明标识符，这样，用于两个序列的范围变量就能用于后面的子句，并且创建了一个匿名类型，简化了对resultSelector参数使用的映射。

然而，如果查询表达式的下一部分是select子句，那么select子句的投影就直接作为resultSelector参数——当你可以一步完成这些转换的时候，创建元素对，然后调用Select是没有意义的。你仍然可以把它看做是“select”步骤所跟随的“join”步骤，尽管两者都被压缩到了一个单独的方法调用中。在我看来，这样在思维模式上更能保持一致，而且这种思维模式也容易理解。除非你打算研究生成的代码，不然可以忽略编译器为你完成的这些优化。

令人高兴的是，在学懂了内连接的相关知识后，下一种连接类型就很容易理解了。

11.5.2　使用join...into子句进行分组连接

我们之前已经看到了来自于普通join子句的结果序列包含成对的元素，其中每个元素来自于各自的输入序列。分组连接（group join）的查询表达式看上去与之类似，不过却具有完全不同的结果。分组连接结果中的每个元素由左边序列（使用它的原始范围变量）的某个元素和右边序列的所有匹配元素的序列组成。后者用一个新的范围变量表示，该变量由join子句中into后面的标识符指定。

我们来把之前的例子改为使用分组连接。代码清单11-13再次显示了所有缺陷和每个缺陷所需的通知，但将通知按“每个缺陷”进行了分解。特别值得注意的是，我们如何用嵌套的foreach循环打印结果。

代码清单11-13　使用分组连接把缺陷和订阅连接到一起

```
var query = from defect in SampleData.AllDefects
            join subscription in SampleData.AllSubscriptions
               on defect.Project equals subscription.Project
               into groupedSubscriptions
            select new { Defect = defect,
                         Subscriptions = groupedSubscriptions };

foreach (var entry in query)
{
   Console.WriteLine(entry.Defect.Summary);
   foreach (var subscription in entry.Subscriptions)
   {
      Console.WriteLine ("  {0}", subscription.EmailAddress);
   }
}
```

每个条目的Subscriptions属性是一个内嵌序列，该序列包含了匹配该数据项缺陷的所有订阅。图11-7显示了两个原始的序列是如何连接到一起的。

内连接和分组连接之间的一个重要差异（即分组连接和普通分组之间的差异）是，对于分组连接来说，在左边序列和结果序列之间是一对一的对应关系，即使左边序列中的某些元素在右边序列中没有任何匹配的元素，也无所谓。这是非常重要的，有时会用于模拟SQL的*左外连接*。在左边元素不匹配任何右边元素的时候，嵌入序列就是空的。与内连接一样，分组连接要对右边序列进行缓冲，而对左边序列进行流处理。

代码清单11-14显示这样一个例子，计算5月份每一天创建的缺陷总数。它使用DateTimeRange类型来作为左边序列，投影则是在一个分组连接结果中的嵌入序列上调用Count()[①]。

① 这只是一个用于示例的简单实现，并不是一个成熟通用的时间范围。

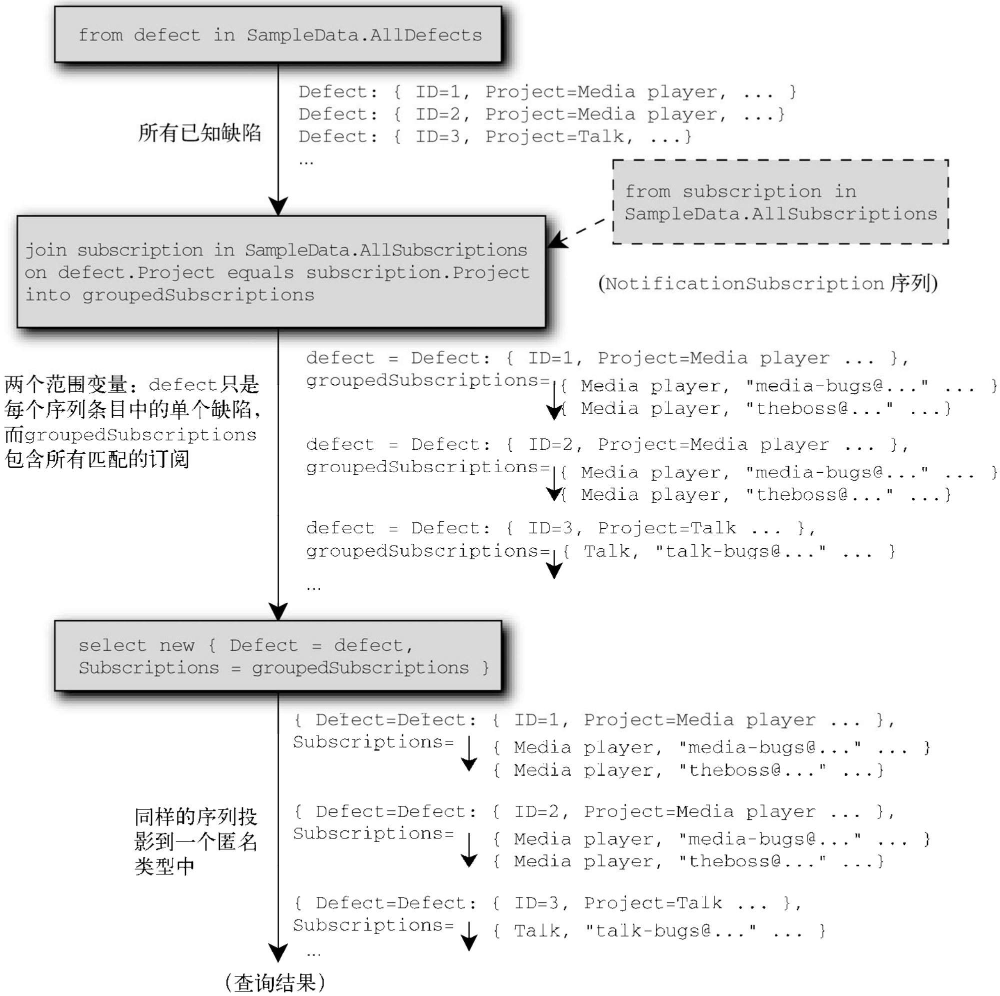


图11-7　代码清单11-13分组连接中的序列。短箭头表示结果条目中内嵌的序列。在输出结果中，某些条目包含同一个缺陷所对应的多个电子邮件地址

代码清单11-14　计算5月份每天产生的缺陷数量

```
var dates = new DateTimeRange(SampleData.Start, SampleData.End);

var query = from date in dates
            join defect in SampleData.AllDefects
               on date equals defect.Created.Date
               into joined
            select new { Date = date, Count = joined.Count() };
foreach (var grouped in query)
{
   Console.WriteLine("{0:d}: {1}", grouped.Date, grouped.Count);
}
```

Count()本身使用立即执行的模式，就是遍历它被调用的序列中的所有元素，不过我们只能在查询表达式的投影部分调用它，所以它就成为Lambda表达式的一部分。这意味着，我们依然可以实现延迟执行：直到开始执行foreach循环才会进行求值。

下面是代码清单11-14执行结果的第一部分，显示了5月第一周每天创建缺陷的数量：

```
05/01/2013: 1
05/02/2013: 0
05/03/2013: 2
05/04/2013: 1
05/05/2013: 0
05/06/2013: 1
05/07/2013: 1
```

编译器将分组连接转译为简单地调用GroupJoin方法，就像Join一样。Enumerable.GroupJoin的签名如下：

```
static IEnumerable<TResult> GroupJoin<TOuter,TInner,TKey,TResult>(
   this IEnumerable<TOuter> outer,
   IEnumerable<TInner> inner,
   Func<TOuter,TKey> outerKeySelector,
   Func<TInner,TKey> innerKeySelector,
   Func<TOuter,IEnumerable<TInner>,TResult> resultSelector
)
```

这个方法签名和内连接的方法签名非常相似，只不过resultSelector参数必须要用于处理右边元素的序列，不能处理单一的元素。同内连接一致，如果分组连接后面紧跟着select子句，那么投影就用作GroupJoin调用的结果选择器，否则，就引入一个透明标识符。在这个例子中，分组连接之后紧接着select子句，所以转译后的查询如下：

```
dates.GroupJoin(SampleData.AllDefects,
                date => date,
                defect => defect.Created.Date,
                (date, joined) => new  { Date = date,
                                         Count = joined.Count() });
```

我们最后要讨论的连接类型称为*交叉连接*（cross join），不过它并不像最初看起来那么直观易懂。

11.5.3　使用多个from子句进行交叉连接和合并序列

到目前为止，我们学到的连接都是相等连接（equijoin）——左边序列中的元素和右边序列要进行匹配。交叉连接不在序列之间执行任何匹配操作：结果包含了所有可能的元素对。它们可以简单地使用两个（或多个）from子句来实现。为便于理解，现在我们只考虑两个from子句的情况——涉及多个from子句时，其实可认为是在前两个from子句上执行交叉连接，接着把结果序列和下一个from子句再次进行交叉连接，以此类推。每个额外的from子句都通过透明标识符添加了自己的范围变量。

代码清单11-15实际演示了一个简单的（但无用的）交叉连接，它生成了一个序列，其中每个数据项都由一个用户和一个项目组成。我故意选取两个完全无关的原始序列，来显示它们没有执行任何匹配。

代码清单11-15 用户和项目的交叉连接

```
var query = from user in SampleData.AllUsers
            from project in SampleData.AllProjects
            select new { User = user, Project = project };
foreach (var pair in query)
{
   Console.WriteLine("{0}/{1}",
                     pair.User.Name,
                     pair.Project.Name);
}
```

代码清单11-15的输出的开头类似于下面这样：

```
Tim Trotter/Skeety Media Player
Tim Trotter/Skeety Talk
Tim Trotter/Skeety Office
Tara Tutu/Skeety Media Player
Tara Tutu/Skeety Talk
Tara Tutu/Skeety Office
```

图11-8显示用于获得这个结果的序列。

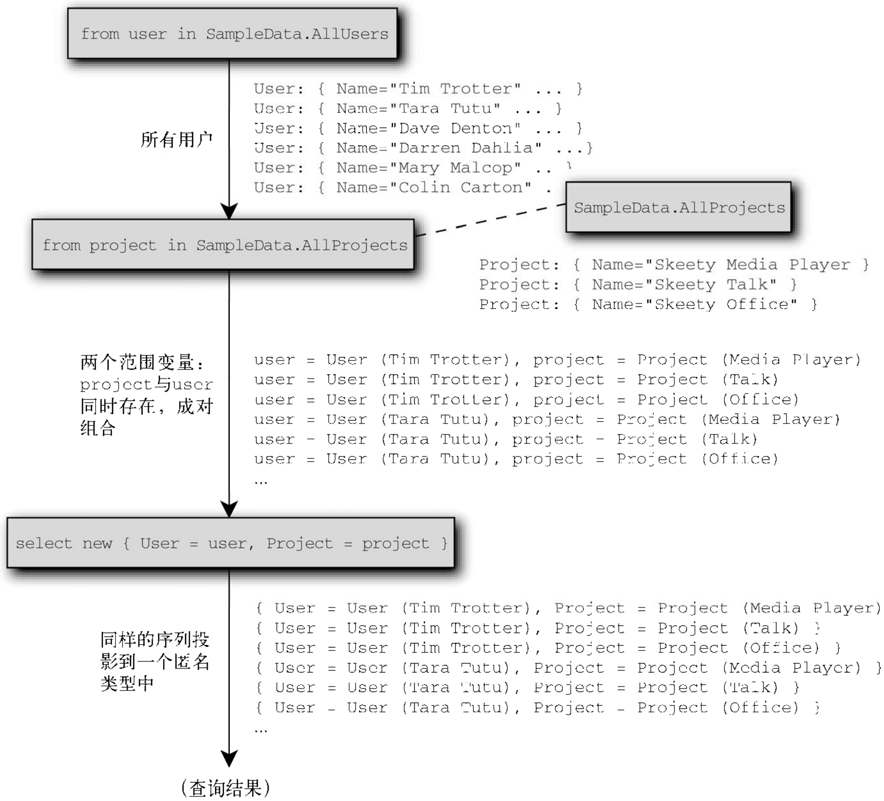


图11-8 代码清单11-15中的序列，对用户和项目进行交叉连接。所有可能的组合都出现在了返回结果中

如果你对SQL比较熟悉，那么到目前为止你应该还感觉比较舒服——它就像指定了多表查询的笛卡儿积。实际上，大多数时候，它正是交叉连接的用法。然而，在你需要的时候，就能发现它更强大：在任意特定时刻使用的右边序列依赖于左边序列的“当前”值。也就是说，左边序列中的每个元素都用来生成右边的一个序列，然后左边这个元素与右边新生成序列的每个元素都组成一对。这并不是通常意义上的交叉连接，而是将多个序列高效地合并（flat）成一个序列。不管我们是否使用真正的交叉连接，查询表达式的转译是相同的，所以为了理解转译过程，我们需要研究一下更复杂的情形。

在深入细节之前，我们来看一下它产生的效果。代码清单11-16显示了一个使用整数序列的简单例子。

代码清单11-16　右边序列依赖于左边元素的交叉连接

```
var query = from left in Enumerable.Range(1, 4)
            from right in Enumerable.Range(11, left)
            select new { Left = left, Right = right };

foreach (var pair in query)
{
   Console.WriteLine("Left={0}; Right={1}",
                     pair.Left, pair.Right);
}
```

代码清单11-16由一个简单整数范围值（1~4）作为开始。我们为其中的每个整数创建了另外一个范围，从11开始，包含同原始整数范围中同样多的元素。通过使用多个`from`子句，左边序列被连接到生成的右边序列的每个元素，输出结果如下：

```
Left=1; Right=11
Left=2; Right=11
Left=2; Right=12
Left=3; Right=11
Left=3; Right=12
Left=3; Right=13
Left=4; Right=11
Left=4; Right=12
Left=4; Right=13
Left=4; Right=14
```

编译器用来生成这个序列所调用的方法是`SelectMany`。它使用单个的输入序列（以我们的说法就是*左边序列*），一个从左边序列任意元素上*生成*另外一个序列的委托，以及一个生成结果元素（其包含了每个序列中的元素）的委托。下面是这个方法的签名，再次写成`Enumerable.SelectMany`的实例方法：

```
static IEnumerable<TResult> SelectMany<TSource,TCollection,TResult>(
   this IEnumerable<TSource> source,
   Func<TSource,IEnumerable<TCollection>> collectionSelector,
   Func<TSource,TCollection,TResult> resultSelector
)
```

和其他连接一样，如果查询表达式中连接操作后面紧跟的是`select`子句，那么投影就作为最后的实参；否则，引入一个透明标识符，从而使左右序列的范围变量在后续查询中都能被访问。

为了更具体一些，下面是代码清单11-16的查询表达式转译后的代码：

```
Enumerable.Range(1, 4)
          .SelectMany(left => Enumerable.Range(11, left),
                      (left, right) => new {Left = left, Right = right})
```

SelectMany的一个有意思的特性是，执行完全是流式的——一次只需处理每个序列的一个元素，因为它为左边序列的每个不同元素使用最新生成的右边序列。把它与内连接和分组连接进行比较，就能看出：在开始返回任何结果之前，它们都要完全加载右边序列。你应该在心中谨记如下问题：序列的预期大小，以及计算多次可能的资源开销，何时考虑要使用哪种类型的连接，哪个作为左边序列，哪个作为右边序列。

SelectMany的合并行为是非常有用的。例如，你可能需要处理大量日志文件，每次处理一行。几乎不用花太多力气，我们就能处理所有行的无缝序列。在可下载的源代码中有下面伪代码的完整版本，其完整的含义和有效性已经非常清晰：

```
var query = from file in Directory.GetFiles(logDirectory, "*.log")
            from line in ReadLines(file)
            let entry = new LogEntry(line)
            where entry.Type == EntryType.Error
            select entry;
```

在短短的5行代码中，我们检索、解析并过滤了整个日志文件集，返回了表示错误的日志项的序列。至关重要的是，我们不会一次性向内存加载单个日志文件的全部内容，更不会一次性加载所有文件——所有的数据都采用流式处理。

在学习了连接后，我们需要学习的最后一项内容就稍微容易理解一些了。我们要学习使用键来对元素进行分组，并在group...by或select子句之后延续查询表达式。

11.6 分组和延续

通过序列中元素的某个属性来对元素进行分组是很常见的需求。在LINQ中可以使用group...by子句轻松做到这一点。在介绍最后这个子句类型的同时，我们也要重新回顾一下最先介绍的select子句来看看一个称为*查询延续*（query continuations）的特性，该特性可以应用到分组和投影上。让我们以一个简单的分组作为开始。

11

11.6.1 使用**group...by**子句进行分组

分组总体上很直观，LINQ则使得它更加容易。要在查询表达式中对序列进行分组，只需要使用group...by子句，语法如下：

```
group projection by grouping
```

该子句与select子句一样，出现在查询表达式的末尾。但它们的相似之处不止于此：*projection*表达式和select子句使用的投影是同样的类型。只不过生成的结果稍有不同。

*grouping*表达式通过其键来决定序列如何分组。整个结果是一个序列，序列中的每个元素本身就是投影后元素的序列，还具有一个Key属性，即用于分组的键；这样的组合是封装在

IGrouping<TKey,TElement>接口中的，它扩展了IEnumerable <TElement>。同样，如果你想根据多个值来进行分组，可以使用一个匿名类型作为键。

我们来看一个有关SkeetySoft缺陷系统的简单例子：基于缺陷的当前分配者来进行分组。代码清单11-17用非常简单的投影形式完成了这个工作，所以结果序列就用分配者表示键，而缺陷序列嵌入到每个条目中。

代码清单11-17　用分配者来分组缺陷——无比简单的投影

```
var query = from defect in SampleData.AllDefects
            where defect.AssignedTo != null
            group defect by defect.AssignedTo;

foreach (var entry in query)
{
   Console.WriteLine(entry.Key.Name);
   foreach (var defect in entry)
   {
      Console.WriteLine("  ({0}) {1}",
                        defect.Severity, defect.Summary);
   }
   Console.WriteLine();
}
```

❶ 过滤未分配的缺陷
❷ 用分配者来分组
❸ 使用每个条目的键：分配者
❹ 遍历数据条目的子序列

代码清单11-17在每日构建报告中是非常有用的，可以快速地看到每个人需要负责哪些缺陷。我们对所有缺陷进行了过滤，排除了那些无须关注的缺陷❶，并用AssignedTo属性进行分组。虽然此刻我们使用的是属性，实际上可以使用任何你想用的东西作为分组表达式——它会应用到序列的每个条目上，然后序列将根据表达式的结果进行分组。注意，分组无法对结果进行流处理，它会对每个元素应用键选择和投影，并缓冲被投影元素的分组序列。尽管它不是流式的，但执行仍然是延迟的，直到开始获取其结果。

应用于分组的投影❷很简单，它只是选择了原始的元素。在我们遍历结果序列时，每个条目都具有一个Key属性，它的类型是User❸，每个条目也都实现了IEnumerable<Defect>，是分配给该用户的缺陷序列❹。

代码清单11-17输出结果的前几行如下所示：

```
Darren Dahlia
  (Showstopper) MP3 files crash system
  (Major) Can't play files more than 200 bytes long
  (Major) DivX is choppy on Pentium 100
  (Trivial) User interface should be more caramelly
```

在Darren的所有缺陷都打印出来后，我们会看到Tara的，接着是Tim的，以此类推。这个实现实际上保留了一个之前看到的分配者列表，并在每次需要的时候都添加一个新的分配者。图11-9显示了贯穿查询表达式的所有生成序列，它们能使执行的顺序更加清晰。

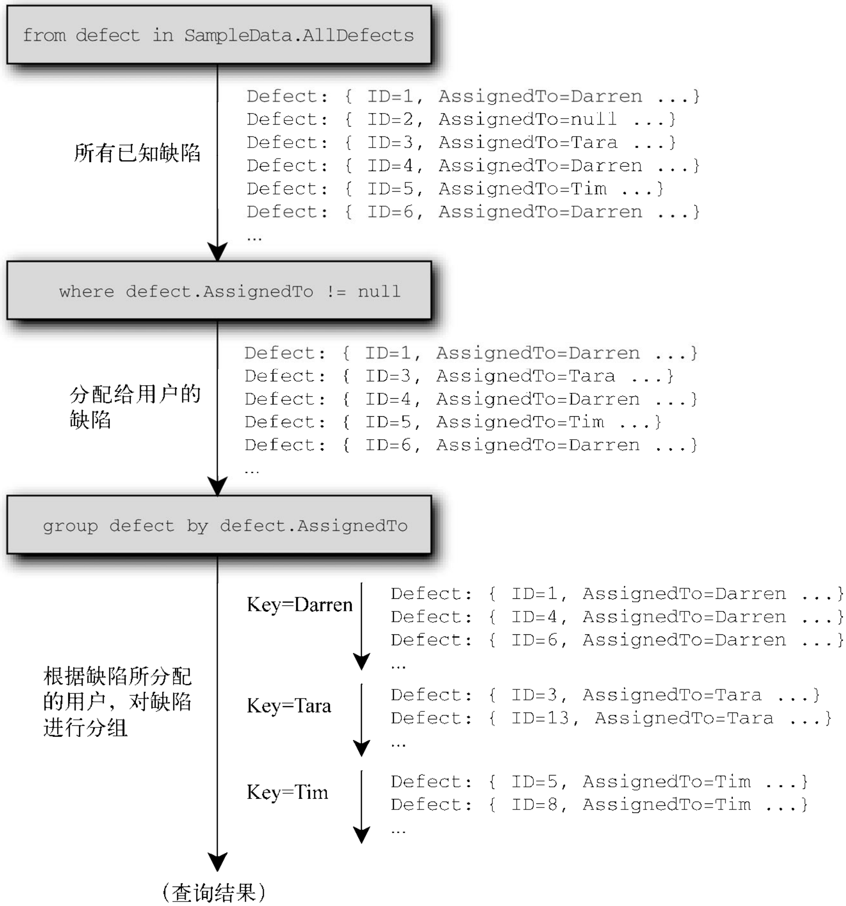


图11-9 按分配者分组缺陷时使用的序列。结果中的每个条目都具有一个Key属性，并且同时还是一个缺陷项的序列

在每个条目的子序列中，缺陷的顺序与原始缺陷序列的顺序是一致的。如果你想改变顺序，可以考虑在查询表达式中显式地设定它，这样会更具可读性。

运行代码清单11-17，可以看到Mary Malcop根本未出现在结果中，因为还没有将任何缺陷分配给她。如果你打算生成一个由用户以及分配给他们的缺陷组成的完整列表，可以使用代码清单11-14中那样的分组连接。

编译器总是对分组子句使用GroupBy的方法调用。当分组子句中的投影很简单的时候——换句话说，在原始序列中的每个数据项都直接映射到子序列中的同一个对象的时候，编译器将使用简单的重载版本（只以分组表达式为参数），它知道如何把每个元素映射到键上。例如，代码清单11-17中的查询表达式会转译为：

```
SampleData.AllDefects.Where(defect => defect.AssignedTo != null)
                     .GroupBy(defect => defect.AssignedTo)
```

在投影不是那么简单的时候，就会使用稍微复杂一点的版本。代码清单11-18给出了一个投影的例子，此时我们只能得到每个缺陷的概要，而不是Defect对象本身。

代码清单11-18　按分配者来分组缺陷——投影只保留概要信息

```
var query = from defect in SampleData.AllDefects
            where defect.AssignedTo != null
            group defect.Summary by defect.AssignedTo;

foreach (var entry in query)
{
   Console.WriteLine(entry.Key.Name);
   foreach (var summary in entry)
   {
      Console.WriteLine("  {0}", summary);
   }
   Console.WriteLine();
}
```

我用粗体标出了代码清单11-18和代码清单11-17之间的不同之处。由于我们投影的是缺陷的概要，所以每个条目中内嵌的序列为IEnumerable<string>。在这个例子中，编译器使用GroupBy的一个重载版本，它用另一个参数来表示投影。代码清单11-18中的查询表达式被转译为如下的表达式：

```
SampleData.AllDefects.Where(defect => defect.AssignedTo != null)
                     .GroupBy(defect => defect.AssignedTo,
                              defect => defect.Summary)
```

分组子句虽然相对简单，却相当有用。即使在我们的缺陷跟踪系统中，你也能很容易就想到，除了在这些例子中使用的分配者外，还可以使用项目、创建者、严重程度或状态来进行分组。

到目前为止，我们的查询表达式以至于整个表达式都是以select或group...by子句作为结尾。而有些时候，你也许打算对结果进行更多的处理——此时，就可以用到查询延续。

11.6.2　查询延续

查询延续提供了一种方式，把一个查询表达式的结果用作另外一个查询表达式的初始序列。它可以应用于group...by和select子句上，语法对于两者是一样的——你只需使用上下文关键字into，并为新的范围变量提供一个名称就可以了。范围变量接着能用在查询表达式的下一部分。

C#规范在解释这个术语时，将它从这种形式

```
first-query into identifier
second-query-body
```

转译为

```
from identifier in (first-query)
second-query-body
```

举个例子也许会更清楚一些。让我们回到用分配者进行缺陷分组的例子中，不过这次我们只

需要分配给每个人的缺陷的数量。我们不在分组子句中用投影来完成，因为那只是应用到了每个独立的缺陷上。我们打算对包含分配者和缺陷序列的分组进行投影，最终的单个元素由分组中的分配者及其分配的缺陷数量组成。这可以通过代码清单11-19来完成。

代码清单11-19　使用另外一个投影来延续分组结果

```
var query = from defect in SampleData.AllDefects
            where defect.AssignedTo != null
            group defect by defect.AssignedTo into grouped
            select new { Assignee = grouped.Key,
                         Count = grouped.Count() };

foreach (var entry in query)
{
   Console.WriteLine("{0}: {1}",
                     entry.Assignee.Name, entry.Count);
}
```

对查询表达式修改的地方用粗体标出。我们能在查询的第二部分使用grouped范围变量，不过defect范围变量不再可用——你可以认为它已经超出作用域了。我们的投影简单地创建了一个包含Assignee和Count属性的匿名类型，使用每个分组的键作为分配者[①]，并计算每个分组中缺陷序列包含的元素个数[②]。

代码清单11-19的结果如下：

```
Darren Dahlia: 14
Tara Tutu: 5
Tim Trotter: 5
Deborah Denton: 9
Colin Carton: 2
```

按照规范，代码清单11-19的查询表达式被转译为如下形式：

```
from grouped in (from defect in SampleData.AllDefects
                 where defect.AssignedTo != null
                 group defect by defect.AssignedTo)
select new { Assignee = grouped.Key, Count = grouped.Count() }
```

接着又执行余下的转译，结果就是如下代码：

```
SampleData.AllDefects
          .Where(defect => defect.AssignedTo != null)
          .GroupBy(defect => defect.AssignedTo)
          .Select(grouped => new { Assignee = grouped.Key,
                                   Count = grouped.Count() })
```

理解延续的另一种方式是，可以把它们看做两个分开的语句。从实际编译器转译的角度看，这不够准确，不过我发现这样可以更容易明白所发生的事情。在这个例子中，查询表达式（以及对query变量进行分配的表达式）可以看做是如下两个语句：

① 即给Assignee属性赋值。——译者注

② 得到的值保留在Count属性中。——译者注

```
var tmp = from defect in SampleData.AllDefects
          where defect.AssignedTo != null
          group defect by defect.AssignedTo;

var query = from grouped in tmp
            select new { Assignee = grouped.Key,
                         Count = grouped.Count() };
```

当然，如果你发现这样更容易阅读的话，也可以在源代码中把原始表达式分解为这样的形式。由于延迟执行的原因，在你开始逐一地访问结果之前，不会执行任何计算。

> **说明　join ... into不是延续**　你很容易掉进这样的陷阱，即看到了上下文关键字into，就认为这是查询延续。对于连接来说，这是不对的。用于分组连接的join ... into子句不能形成一个延续的结构。主要的区别在于，在分组连接中，你仍然可以使用所有的早期范围变量（用于连接右边名称的范围变量除外）。而对比本节的查询不难发现，延续会清除之前的范围变量，只有在延续中声明的范围变量才能在供后续使用。

让我们再来扩展一下这个例子，看看多个延续如何使用。目前我们的结果是没有排序的——那么就来改变一下，以便能看出谁分配到的缺陷最多。我们可以在第1个延续之后使用let子句，不过代码清单11-20显示了另一种方法，在我们当前的表达式后面使用第2个延续。

代码清单11-20　在group和select子句之后的查询表达式延续

```
var query = from defect in SampleData.AllDefects
            where defect.AssignedTo != null
            group defect by defect.AssignedTo into grouped
            select new { Assignee = grouped.Key,
                         Count = grouped.Count() } into result
            orderby result.Count descending
            select result;

foreach (var entry in query)
{
   Console.WriteLine("{0}: {1}",
                     entry.Assignee.Name,
                     entry.Count);
}
```

代码清单11-19和代码清单11-20之间的变化用粗体标出。由于我们已得到相同类型的序列，所以不必改变任何输出代码——我们只需对它进行排序。

这次，转译后的查询表达式如下：

```
SampleData.AllDefects
          .Where(defect => defect.AssignedTo != null)
          .GroupBy(defect => defect.AssignedTo)
          .Select(grouped => new { Assignee = grouped.Key,
                                   Count = grouped.Count() })
          .OrderByDescending(result => result.Count);
```

它和我们在10.3.6节遇到的第1个缺陷跟踪查询是如此地相似，这纯属巧合。最后的select

子句实际上什么都没做，所以C#编译器忽略了它。然而，由于所有查询表达式必须以select或group...by子句来结尾，所以在查询表达式中必须包含它。而且，你可以自由地针对延续查询使用不同的投影或执行其他操作——连接、进一步的分组，等等。在查询表达式不断增长的同时，要时刻注意它的可读性。

谈到可读性，当你编写LINQ查询时有多种选择。

11.7　在查询表达式和点标记之间作出选择

正如我们在本章看到的，查询表达式在编译之前，先被转译成普通的C#。用普通的C#调用LINQ查询操作符来代替查询表达式，这种做法并没有官方名称，很多开发者称其为点标记（dot notation）[①]。每个查询表达式都可以写成点标记的形式，反之则不成立：很多LINQ操作符在C#中不存在等价的查询表达式。最重要的问题是：什么时候使用哪种语法？

11.7.1　需要使用点标记的操作

最明显的必须使用点标记的情形是调用Reverse、ToDictionary这类没有相应的查询表达式语法的方法。然而即使查询表达式支持你要使用的查询操作符，也很有可能无法使用你想使用的特定重载。

例如，Enumerable.Where包含一个重载，将父序列的索引作为另一个参数传入委托。因此，要从序列中排除其他项可以这样：

```
sequence.Where((item, index) => index % 2 == 0)
```

Select也有类似的重载，因此，如果你要在排序之后获取序列排序之前的索引，可以这样：

```
sequence.Select((Item, Index) => new { Item, Index })
        .OrderBy(x => x.Item.Name)
```

该示例还展示了另外一种选择，我们不妨借鉴：在匿名类型中直接使用Lambda表达式参数，可以打破参数名首字母使用小写字母的惯例，然后使用投影初始化器，就可以避免编写new { Item = item, Index = index }这样让人注意力分散的代码。当然，你也可以忽略属性名称的惯例，让匿名类型的属性以小写字母开头（如item和index）。如何选择完全取决于你，并且都值得一试。尽管一致性通常很重要，但在这里却无足轻重，因为不一致性带来的影响仅仅局限在所讨论的方法内：你不会在公共API中公开这些名称，也不会在其余的类中使用。

很多查询操作符还支持自定义比较器，最常见的用途是排序和连接。在我看来，它们并不经常使用，但偶尔会非常有价值。例如，你需要以不区分大小写的方式根据人名执行一个连接，可以将StringComparer.OrdinalIgnoreCase（或特定区域性比较器）指定为Join方法的最后一个参数。再次强调，如果你觉得某个操作符几乎实现了你想要的，但还没有尽善尽美，可以检查一下文档看看有没有其他重载。

11

① 从现在开始我将使用这个术语，如果你听到有人在谈论流畅的标记（fluent notation），所指的为同一个东西。

当你被迫使用点标记时，可以很容易地下决心使用它，但如果遇到也可以使用查询表达式的情形呢？

11.7.2　使用点标记可能会更简单的查询表达式

有些开发者会在能使用查询表达式的地方都使用查询表达式。就我个人来说，我会看看所做的是什么样的查询，并判断哪一种方法更具可读性。

例如下面的查询表达式，与本章开头所使用的十分类似：

```
var adults = from person in people
             where person.Age >= 18
             select person;
```

这3行代码显得累赘，因为它所做的仅仅是过滤。这时我倾向于使用点标记：

```
var adults = people.Where(person => person.Age >= 18);
```

这样显得更清晰，每一部分都包含我们感兴趣的内容。

还有一种情况，使用点标记可以比查询表达式会更加清晰，就是你被迫要在查询的某一部分使用点标记。例如，你要使用`ToList()`扩展方法返回成人姓名列表。（我在本例中也使用了投影，这样比较显得更加公允。）查询表达式如下：

```
var adultNames = (from person in people
                  where person.Age >= 18
                  select person.Name).ToList();
```

点标记的等价形式为：

```
var adultNames = people.Where(person => person.Age >= 18)
                       .Select(person => person.Name)
                       .ToList();
```

在我看来，需要使用括号的查询表达式有些丑陋。这只是个人喜好问题，本节只是想让你意识到还*存在*另一种选择。如果你想更好地使用LINQ，就应该适应这两种标记方法，根据查询本身来选择某种风格没有什么不好的。正如我们所见，生成的代码是完全等价的。当然，这并不是说我不喜欢查询表达式。

11.7.3　选择查询表达式

在介绍了一些点标记占优势的情形之后，我不得不指出，在执行某些操作时（特别是连接操作），如果查询表达式使用了透明标识符，这时点标记的可读性就没那么高了。透明标识符之美在于它们是透明的，甚至你只看查询表达式的话都看不到它们！即使一个简单的`let`子句可能就足以让你选择查询表达式：如果引入一个新的匿名类型只是为了在查询中扩充上下文，那么很快就会让你产生厌烦情绪。

查询表达式占优势的另一种情况是，需要多个Lambda表达式，或多个方法调用。这同样也包括连接在内，你需要为每个连接方指定键选择器，以及最终的结果选择器。例如，以下是我之前介绍内连接查询的缩减版：

```
from defect in SampleData.AllDefects
join subscription in SampleData.AllSubscriptions
   on defect.Project equals subscription.Project
select new { defect.Summary, subscription.EmailAddress }
```

在IDE中，你可以把整个join子句放在一行，这可以增加可读性。然而其点标记版本则让人不寒而栗：

```
SampleData.AllDefects.Join(SampleData.AllSubscriptions,
   defect => defect.Project,
   subscription => subscription.Project,
   (defect, subscription) => new { defect.Summary,
                                   subscription.EmailAddress })
```

最后一个参数在IDE中也可以只占用一行，但仍然很丑陋，因为这些Lambda表达式没有上下文信息：你无法立即说出每个参数的含义。C# 4的命名实参 虽然有助于理解，却使查询更加臃肿。

用点标记做复杂排序同样不是很好，想想看你是想阅读这样的orderby子句：

```
orderby item.Rating descending, item.Price, item.Name
```

还是这样的三个方法调用：

```
.OrderByDescending(item => item.Rating)
.ThenBy(item => item.Price)
.ThenBy(item => item.Name)
```

在查询表达式中修改这些排序的优先级是很简单的，只需要交换位置即可。但在点标记中，你可能还需要将OrderBy转换为ThenBy，或反之。

需要重申的是，我并没有尝试将个人偏好强加到你的代码上。我只是想让你知道哪些是可用的，并在做决定前仔细斟酌。当然，这只是编写易读代码的一个方面，但对C#来说，却是一个全新的领域。

11.8 小结

在本章中，我们看到了LINQ to Objects如何和C# 3进行交互，然后详细分析了查询表达式如何先被转译为不涉及查询表达式的代码，接着以常规方式进行编译。我们看到所有的查询表达式都会形成一系列序列，并应用了在每一步上都有一些描述的转译。在很多情况下，这些序列都使用延迟执行来进行计算，即只有在第一次访问的时候才获取数据。

与其他所有的C# 3特性相比，查询表达式看上去有点另类——它更像SQL语句，而不是我们习惯的C#代码。它们看上去如此古怪的一个原因是，它们是*声明式*的，而不是*命令式*的——查询关心的是最后的结果，而不是完成查询的各个步骤。这更多地与函数式编程的思想联系到一起。确实需要一定的时间才能把握这个东西，而且也不一定适合所有情况，在适合使用声明式语法的地方，它极大地提高了可读性，并让代码更容易测试，也更容易进行并行化处理。

不要愚蠢地认为，LINQ只能用于数据库：内存中普通的集合的操作也是很常见的，正如我们所看到的那样，查询表达式和Enumerable中的扩展方法对其支持得很好。

从真正的意义上说，你目前已经看到了C# 3的所有新特性了！虽然我们尚未看到任何其他LINQ提供器，不过在用它来处理XML和SQL的时候，对编译器为我们完成的事情肯定会有一个更清楚的认识。编译器本身并不知道LINQ to Objects、LINQ to SQL或任何其他的提供器之间的区别：它只是不假思索地遵循同样的规则。

在下一章中我们将来看看，这些规则构成了“LINQ拼图游戏”的最后一块拼图——为了让查询表达式的各种子句能在不同的平台上执行，它们会把Lambda表达式转换为表达式树。同时我们还将看到LINQ功能的其他一些示例。

第 12 章 超越集合的LINQ

本章内容

- LINQ to SQL
- IQueryable和表达式树查询
- LINQ to XML
- 并行LINQ
- .NET响应式扩展
- 编写自己的操作符

假设外星人造访，并向你询问什么是“文化”。你该如何在很短的时间内描述人类文化的多样性呢？你很可能会选择向他展示文化，而不是抽象地描述：比如，带他去新奥尔良的爵士乐俱乐部，或到斯卡拉[①]欣赏歌剧，巴黎的卢浮宫也不错，最后再去埃文河畔斯特拉特福[②]看一场正宗的莎士比亚戏剧。

那么，这样的话这个外星人就深谙什么是文化了吗？他能作曲、著书、跳芭蕾、造雕塑吗？显然不能。但他会对文化有一个感性认识——它丰富多彩，点亮了人们的生活。

本章的目的亦是如此。我们已经介绍了C# 3的所有特性，但对于LINQ，还没有提供充分的环境能让你真正地领会它的奥妙之处。在本书第1版发布时，LINQ技术还没有得到充分地应用，但现在已是百花齐放。有来自微软的实现，也有第三方的，这并不出乎我的意料，但我却着实被这些技术的不同性质所深深吸引。

我们将分别通过示例介绍LINQ展现其自身的不同方式。我主要演示微软的技术，因为它们最具代表性。这并不是说在LINQ生态系统中第三方技术不受欢迎，实际上有大量的商业和开源项目，可以访问不同的数据源，或在已有提供器的基础上构建额外的特性。

与本书其他部分不同，我们对这些话题都是蜻蜓点水般的介绍。这里的重点不是学习它们的细节，而是让你领会LINQ的精神。要深入研究这些技术，我建议你仔细阅读专门的书籍或相关文档。我没有在每节的最后都写上“要了解更多有关LINQ to [xxx]的内容，请参考……”，但真

① 斯卡拉（La Scala），意大利著名的歌剧院。——译者注

② 埃文河畔斯特拉特福（Stratford-upon-Avon）是英格兰中部的小镇，是英国著名剧作家莎士比亚的故乡。——译者注

的有必要去看一看。这里的每一项技术都具备很多超越查询的功能，但我还是会将注意力集中在直接与LINQ相关的领域内。

我们首先介绍的这个提供器，是在LINQ刚引入时就受到广泛关注的LINQ to SQL。

12.1　使用 LINQ to SQL 查询数据库

你肯定已经知道了LINQ to SQL可以将查询表达式转换为SQL，然后在数据库中执行。实际上不止如此，它还是一个完整的ORM解决方案。不过，我们将只关注LINQ to SQL的查询方面，而不会涉及ORM需要处理的并发等细节。我将介绍刚好够用的知识，这样你就可以亲自动手实践了。本书网站（http://csharpindepth.com）上有相关的数据库和代码。数据库为SQL Server 2005格式，这样即使你没有安装最新版的SQL Server，也能轻松地玩转LINQ。当然，微软确保了LINQ to SQL可以用于最新版的SQL Server。

说明　**为什么是LINQ to SQL，而不是Entity Framework？**　说到"新版"，你可能会问为什么我选择了LINQ to SQL而不是Entity Framework（现在是微软推荐的解决方案，同样支持LINQ）。答案仅仅是简单：Entity Framework毫无疑问在多个方面都比LINQ to SQL强大，但是它包含太多其他概念，这些概念解释起来太占篇幅。我会尽量让你感觉到LINQ所提供的一致性（偶尔不一致），这些对于LINQ to SQL和Entity Framework同样都是适用的。

在编写查询之前，我们需要建立一个数据库和一个在代码中表示数据库的模型。

12.1.1　数据库和模型

LINQ to SQL需要有关数据库的元数据，来知道哪个类与哪个数据库表相对应等信息。可以用几种不同的方式来表示这种元数据，而我这里使用的是Visual Studio内嵌的LINQ to SQL设计器。你可以先设计实体，然后让LINQ创建数据库，也可以先设计数据库，然后让Visual Studio生成实体。它们各有利弊，我个人喜欢第二种方式。

1. 创建数据库架构

把第11章的一些类映射为数据库表是很简单的。每个表有一个带有适当名称的自增整数ID列：`ProjectID`、`DefectID`，等等。表之间的引用仅使用同样的名称，比如`Defect`表有一个带有外键约束的`ProjectID`列。

但在这些简单的规则之中还是存在一些例外：

- `User`是T-SQL中的保留字，所以`User`类映射为`DefectUser`表；
- 枚举类型（状态、严重程度和用户类型）没有对应的数据表：它们的值仅映射为`Defect`表和`DefectUser`表中的`tinyint`列；
- `Defect`表具有两个指向`DefectUser`表的链接，一个指向创建缺陷的用户，另一个指向当前的接收人。分别用`CreatedByUserId`和`AssignedToUserId`列来表示。

2. 创建实体类

创建好数据表后，在Visual Studio中生成实体类就非常容易了。只需打开Server Explorer（在View菜单中选择Server Explorer命令），并添加一个指向SkeetySoftDefects数据库的数据源（在Data Connections上单击鼠标右键，选择Add Connection命令）。你应该可以看到4个数据表：Defect、DefectUser、Project和NotificationSubscription。

接下来，你可以在项目中添加一个LINQ to SQL classes类型的新项。所生成的表示整个数据库模型的类的名称，取决于这个新项的名称。我使用`DefectModel`这个名称——因此数据上下文（data context）类被命名为`DefectModelDataContext`。在创建了新项之后，设计器就会打开。

接着，你可以将这4个表从Server Explorer拖曳到设计器中，设计器会推算出所有关联关系。在这之后，你可以调整它们的布局，调整实体的各种属性。下面是我进行修改的地方。

- 把`DefectID`属性重命名为`ID`，以匹配之前的模型。
- 把`DefectUser`表重命名为`User`（所以尽管数据表依然称作`DefectUser`，我还是可以像之前那样，生成一个称作`User`的类）。
- 把`Severity`、`Status`和`UserType`属性的类型修改为它们的等价枚举类型（要把这些枚举类型复制到该项目中）。
- 我修改了用于`Defect`和`DefectUser`之间父子关系的属性名称——设计器可以猜测出其他关联关系的适合名称，不过在这里，由于同一对数据表之间有两个关联关系，所以会出现问题。我将它们取名为`AssignedTo/AssignedDefects`和`CreatedBy/CreatedDefects`。

图12-1显示了在完成所有改变之后的设计器图表。可以看到，除了没有枚举类型，图12-1中的模型和图11-3中的类图完全一样。

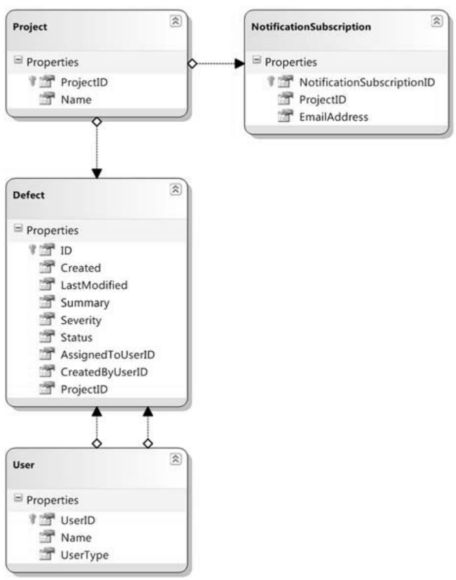


图12-1　LINQ to SQL类设计器显示了重新布局和修改后的实体

12

如果看一下由设计器生成的C#代码（DefectModel.designer.cs），你会发现5个分部类：每个实体1个，还有1个我早前提到的DefectModelDataContext类。将它们设计为分部类很有用，我们可以添加额外的构造函数来匹配之前内存中的类，这样就可以复用第11章中用于创建示例数据的代码，而无须太多更改。简便起见，我没有包含插入代码，但查看了源代码中的PopulateDatabase.cs之后，你应该可以很轻松地进行模仿。当然，你不必这么做，可下载的数据库中已经生成了这些数据。

现在，SQL中具备了架构，C#有了实体模型和一些示例数据，可以开始查询了。

12.1.2　用查询表达式访问数据库

我想你已经猜到要讲的内容了，但愿这样不会让大家对这部分内容失去兴趣。我们将在数据源上执行查询表达式，并观察LINQ to SQL是如何在运行时把查询转换为SQL的。为了让大家感觉熟悉，这里将使用在第11章中的内存集合上执行过的某些相同的查询。

1. 第1个查询：查找分配给Tim的缺陷

我将跳过前面那些简单的例子，而从代码清单11-7中查找分配给Tim的未解决缺陷的查询开始。下面列出了用于比较的代码清单11-7中的查询部分：

```
User tim = SampleData.Users.TesterTim;

var query = from defect in SampleData.AllDefects
            where defect.Status != Status.Closed
            where defect.AssignedTo == tim
            select defect.Summary;
```

与代码清单11-7等价的LINQ to SQL代码显示在代码清单12-1中。

代码清单12-1　查询数据库以找出Tim的全部未解决缺陷

```
using (var context = new DefectModelDataContext())        ❶ 创建用于操作的上下文
{
   context.Log = Console.Out;                             ❷ 打开控制台日志

   User tim = context.Users                               ❸ 查询数据库来找出Tim
                     .Where(user => user.Name == "Tim Trotter")
                     .Single();

   var query = from defect in context.Defects             ❹ 查询数据库找出Tim的未解决缺陷
               where defect.Status != Status.Closed
               where defect.AssignedTo == tim
               select defect.Summary;

   foreach (var summary in query)
   {
      Console.WriteLine(summary);
   }
}
```

有必要对代码清单12-1进行详细地解释，因为它是全新的。我们首先新建了一个数据上下文❶。它包括很多功能，可以负责创建连接和事务管理、查询转译、跟踪实体的变化和处理一致性。就

本章而言，我们可以将数据上下文看成与数据库通信的工具。我们不会在此讨论过于深入的特性，只是使用了一个非常好用的功能：让数据上下文将所有执行的SQL命令都输出到了控制台上❷。本节代码中所使用的与模型相关的属性（Defects、Users等），均为Table<T>类型（以相应实体类型为类型参数）。它们是查询的数据源。

我们不能在主查询中使用SampleData.Users.TesterTim以确认Tim的身份，因为对象不知道在DefectUser表中Tim数据行的ID。相反，我们使用一个单独的查询来加载Tim的用户实体❸，我在这里使用了点标记，不过查询表达式也是可以的。在查询表达式结尾的Single方法调用仅返回查询的单个结果，如果一个元素都没有，那么就会抛出一个异常。在真实的情况中，你拥有的实体可能为其他操作（如登录）的结果——即使没有完整的实体，也会有它的ID，这个ID在主查询中的用法也是完全一样的。本例还可以使用另外一种方法，修改未解决缺陷查询，并根据接收人的名字进行过滤。不过这不能很好地体现原查询的精神。

在查询表达式中❹，内存中的查询和LINQ to SQL查询的唯一区别是数据源——我们使用context.Defects而不是SampleData.AllDefects。最后的结果是一样的（虽然顺序是否一致不能保证），但是所有的工作都在数据库上完成。

由于我们让数据上下文记录了生成的SQL，运行代码时就可以看到具体发生了什么。控制台输出显示了在数据库上执行的两个查询，以及查询参数值[①]：

```
SELECT [t0].[UserID], [t0].[Name], [t0].[UserType]
FROM [dbo].[DefectUser] AS [t0]
WHERE [t0].[Name] = @p0
-- @p0: Input String (Size = 11; Prec = 0; Scale = 0) [Tim Trotter]

SELECT [t0].[Summary]
FROM [dbo].[Defect] AS [t0]
WHERE ([t0].[AssignedToUserID] = @p0) AND ([t0].[Status] <> @p1)
-- @p0: Input Int32 (Size = 0; Prec = 0; Scale = 0) [2]
-- @p1: Input Int32 (Size = 0; Prec = 0; Scale = 0) [4]
```

注意，由于我们要填充整个实体，第1个查询提取了用户的所有属性的——不过第2个查询只提取了摘要，因为我们只需要这个。LINQ to SQL还把我们在第2个查询中两个分开的where子句转换为了数据库中单个的过滤条件。

LINQ to SQL能够转译很多表达式。我们来看一下第11章中更复杂的例子，看看生成什么样的SQL。

12

2. 为更复杂的查询生成SQL：let子句

下一个查询展示了，在我们用let子句引入某种“临时变量”的时候所发生的事情。在第11章中，如果你还记得的话，我们考虑了一种非常奇怪的情况——假设计算字符串长度的操作需要花费很长的时间。同样，除了数据源有所不同，我们的查询表达式和代码清单11-11中的查询表达式完全一样。代码清单12-2显示了LINQ to SQL代码。

① 生成的额外日志输出显示了数据上下文的某些细节，为了避免大家从SQL上转移注意力，我删除了它们。当然，控制台输出还包含由foreach循环打印的缺陷摘要。

代码清单12-2　在LINQ to SQL中使用let子句

```
using (var context = new DefectModelDataContext())
{
   context.Log = Console.Out;

   var query = from user in context.Users
               let length = user.Name.Length
               orderby length
               select new { Name = user.Name, Length = length };

   foreach (var entry in query)
   {
      Console.WriteLine("{0}: {1}", entry.Length, entry.Name);
   }
}
```

生成的SQL和我们在图11-5中看到的序列本质非常相似——最里面的序列（对应图11-5中的第1个框）是用户列表。它被转换为了名称/长度对的序列（嵌套的select语句），接着应用一个无操作的投影，并以长度进行排序：

```
SELECT [t1].[Name], [t1].[value]
FROM (
   SELECT LEN([t0].[Name]) AS [value], [t0].[Name]
   FROM [dbo].[DefectUser] AS [t0]
   ) AS [t1]
ORDER BY [t1].[value]
```

这是一个很好的例子，可以说明生成的SQL语句过于啰嗦。尽管在查询表达式上执行排序的时候，我们不能引用最终输出序列的元素，但在SQL中却可以。下面这个较简单的查询可以正常工作：

```
SELECT LEN([t0].[Name]) AS [value], [t0].[Name]
FROM [dbo].[DefectUser] AS [t0]
ORDER BY [value]
```

当然，重要的在于查询优化器在数据库上做了些什么——显示在SQL Server Management Studio Express中的执行计划对于两个查询来说是相同的，所以看上去我们没有损失什么。

我们最后要介绍的LINQ to SQL查询是连接。

12.1.3　包含连接的查询

我们用通知订阅连接项目的例子，来同时尝试内连接和分组连接。我猜想你现在已经习惯不断钻研了——代码的模式对于每个查询都是一样的，所以从现在开始，如果没有其他情况出现，我们将只显示查询表达式和生成的SQL。

1. 显式连接：使用通知订阅来匹配缺陷

我们的第一个查询是最简单的连接类型——使用LINQ连接子句的内部相等连接：

```
// 查询表达式（从代码清单11-12修改而来）
from defect in context.Defects
join subscription in context.NotificationSubscriptions
   on defect.Project equals subscription.Project
```

```
select new { defect.Summary, subscription.EmailAddress }

-- Generated SQL
SELECT [t0].[Summary], [t1].[EmailAddress]
FROM [dbo].[Defect] AS [t0]
INNER JOIN [dbo].[NotificationSubscription] AS [t1]
ON [t0].[ProjectID] = [t1].[ProjectID]
```

一点都不奇怪，它在SQL中使用内连接。在这个例子中，非常容易就猜到生成的SQL。那么分组连接会怎么样呢？下面的内容稍微变得有点复杂：

```
// 查询表达式（从代码清单11-13修改而来）
from defect in context.Defects
join subscription in context.NotificationSubscriptions
   on defect.Project equals subscription.Project
   into groupedSubscriptions
select new { Defect = defect, Subscriptions = groupedSubscriptions }

-- Generated SQL
SELECT [t0].[DefectID] AS [ID], [t0].[Created],
[t0].[LastModified], [t0].[Summary], [t0].[Severity],
[t0].[Status], [t0].[AssignedToUserID],
[t0].[CreatedByUserID], [t0].[ProjectID],
[t1].[NotificationSubscriptionID],
[t1].[ProjectID] AS [ProjectID2], [t1].[EmailAddress],
   (SELECT COUNT(*)
    FROM [dbo].[NotificationSubscription] AS [t2]
    WHERE [t0].[ProjectID] = [t2].[ProjectID]) AS [count]
FROM [dbo].[Defect] AS [t0]
LEFT OUTER JOIN [dbo].[NotificationSubscription] AS [t1]
ON [t0].[ProjectID] = [t1].[ProjectID]
ORDER BY [t0].[DefectID], [t1].[NotificationSubscriptionID]
```

最主要的改变就是生成了大量的SQL！有两件重要的事情需要注意。首先，它使用左外连接（left outer join）而非内连接，所以即使没有任何人订阅某项目，我们依旧能看到它的缺陷。如果你想要一个没有进行分组的左外连接，习惯的做法就是使用一个分组连接，并接着使用一个额外的`from`子句，该子句在嵌入序列上使用`DefaultIfEmpty`扩展方法。它看起来确实比较奇怪，不过却可以很好的运行。

第二件奇怪的事情是，前面的查询在数据库中为每个分组计算了总数。这实际上是LINQ to SQL略施小计，以保证所有的处理过程都能在服务器上完成。本机实现不得不在获取所有结果后，在内存中执行分组。在某些情况下，提供器会使用一些技巧来避免计数运算，仅在分组的ID改变时进行标记，不过这种方式对于某些查询会存在问题。有可能，LINQ to SQL以后的实现将能够根据确切的查询来切换执行步骤。

要想在SQL中看到连接，你并不需要在查询表达式中显式地编写它。就算它将会被转换为SQL，我们最后介绍的查询是通过属性访问表达式隐式创建的连接。

2. 隐式连接：显示缺陷概要和项目名称

我们举一个简单的例子。假设我们想列出每个缺陷，显示它的概要，以及所属项目的名称。在这里查询表达式只是个投影问题：

```
// Query expression
from defect in context.Defects
select new { defect.Summary, ProjectName = defect.Project.Name }

-- Generated SQL
SELECT [t0].[Summary], [t1].[Name]
FROM [dbo].[Defect] AS [t0]
INNER JOIN [dbo].[Project] AS [t1]
ON [t1].[ProjectID] = [t0].[ProjectID]
```

注意，我们通过一个属性从缺陷导航到项目——LINQ to SQL把那样的导航转换为内连接。在这里能够使用内连接，是由于架构在`Defect`表的`ProjectID`列上具有非可空约束——每个缺陷都要属于某个项目。然而，不是每个缺陷都有接收人——`AssignedToUserID`字段可以为空，所以如果我们在投影中使用分配者，就会生成一个左外连接：

```
// Query expression
from defect in context.Defects
select new { defect.Summary, Assignee = defect.AssignedTo.Name }

-- Generated SQL
SELECT [t0].[Summary], [t1].[Name]
FROM [dbo].[Defect] AS [t0]
LEFT OUTER JOIN [dbo].[DefectUser] AS [t1]
ON [t1].[UserID] = [t0].[AssignedToUserID]
```

当然，如果你通过更多的属性来导航，产生的连接就会变得越来越复杂。在此我不打算深入细节——重要的事情是LINQ to SQL做了很多查询表达式的分析工作，来得到必要的SQL。要执行这种分析，显然需要能够看到我们指定的查询。

我们先撇开LINQ to SQL这个特例，考虑在一般情况下这种LINQ提供器应该怎么做。这对于所有需要分析查询而不仅仅是处理委托的提供器来说，都是适用的。这就是为什么把表达式树引入C#的原因。现在，终于到了学习这一点的时候了。

12.2　用 `IQueryable` 和 `IQueryProvider` 进行转换

在本节中，我们打算学习如下基础知识：LINQ to SQL如何管理查询表达式到SQL的转换过程。如果你要实现自己的LINQ提供器，这将是一个非常好的起点。（不过不要低估此过程中的技术难度，如果你喜欢挑战，实现LINQ提供器一定很有意思。）这是在本章中理论性最强的一个小节，不过对于想深入了解LINQ是如何决定使用内存中的处理过程、数据库或者其他一些查询引擎时是非常有用的。

我们在LINQ to SQL中看到的所有查询表达式中，数据源都是`Table<T>`。不过，如果你看一下`Table<T>`，你就会发现它没有`Where`、`Select`和`Join`方法，或任何其他的标准查询操作符。但是，它利用了和LINQ to Objects同样的技巧——LINQ to Objects中的数据源总是实现`IEnumerable<T>`（可能在调用`Cast`或`OfType`之后），然后使用`Enumerable`中的扩展方法，而`Table<T>`实现了`IQueryable<T>`并使用`Queryable`的扩展方法。我们将看到LINQ如何构建了表达式树，以及提供器如何在恰当的时候执行表达式对。

首先让我们开始看一下IQueryable<T>由什么构成。

12.2.1 IQueryable<T>和相关接口的介绍

如果你在文档中查找IQueryable<T>，来看它直接包含（而不是继承）的成员，你绝对会失望。没有任何这样的成员。相反，它是从IEnumerable<T>和非泛型的IQueryable继承而来，而IQueryable又继承于非泛型的IEnumerable。那么，此处IQueryable就是那些新颖而激动人心的成员所在的对象吗？对，几乎可以这么说。实际上，IQueryable仅有3个属性：QueryProvider、ElementType和Expression。QueryProvider属性是IQueryProvider类型——另一个需要考虑的新接口。

是不是有点晕头转向了？也许图12-2会有所帮助——这个类图包含了所有直接涉及的接口。

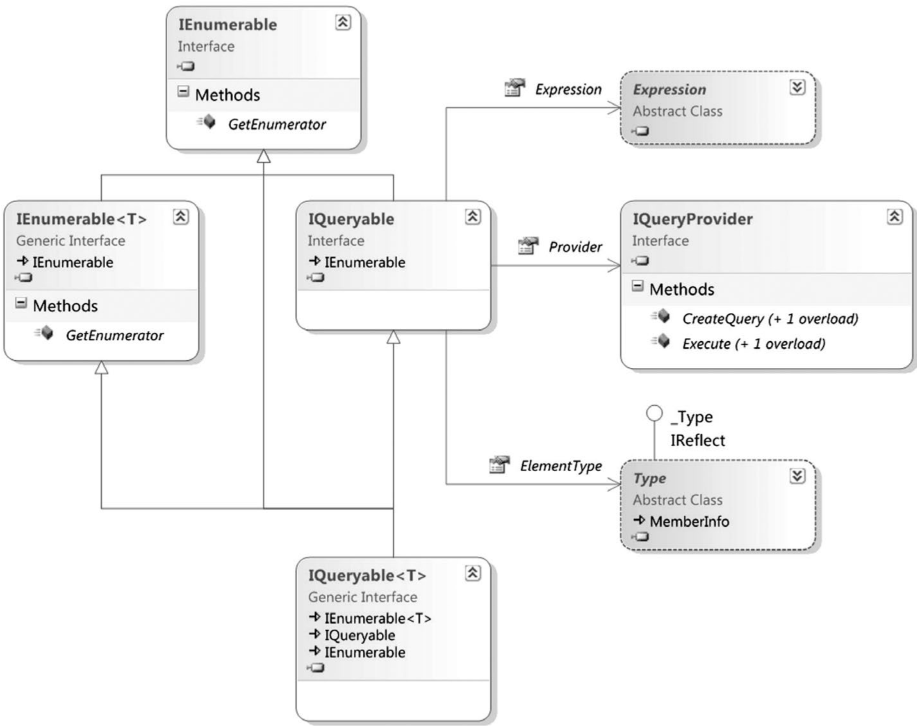


图12-2 IQueryable<T>中所涉及接口的类图

理解IQueryable的最简单方式就是，把它看作一个查询，在执行的时候，将会生成结果序列。从LINQ的角度看，由于是通过IQueryable的Expression属性返回结果，所以查询的详细信息就保存于表达式树中。一个查询进行执行，就是开始遍历IQueryable的过程（换句话说，即调用GetEnumerator方法，然后对其结果调用MoveNext方法），或者调用IQueryProvider上的Execute方法并传递表达式树。

那么，在对IQueryable有了一定的理解后，IQueryProvider又是什么呢？我们不仅用它来完成查询，还用它做更多的事情——我们能用它构建一个更大的查询，这就是LINQ中标准查询操作符的用途[①]。为了构建一个查询，我们需要在相关的IQueryProvider上使用CreateQuery[②]方法。

也可把数据源看作是简单的查询（例如，用SQL编写的SELECT * FROM SomeTable）——调用Where、Select、OrderBy及类似方法会生成不同的查询，具体生成的查询是什么取决于第一个关键字。给定任何IQueryable查询后，你可通过执行如下步骤来创建新的查询：

(1) 请求现有查询的查询表达式树（使用Expression属性）；

(2) 构建一个新的表达式树，包含最初的表达式和你想要的额外功能（例如，过滤、投影或排序）；

(3) 请求现有查询的查询提供器（使用Provider属性）；

(4) 调用提供器的CreateQuery方法，传递新表达式树。

在这些步骤中，唯一的难点就是创建新的表达式树。幸好，在静态Queryable类中有一堆扩展方法可以帮助我们完成。理论知识已经足够——让我们开始来实现接口，以便能实际地看到这些东西。

12.2.2　模拟接口实现来记录调用

先别太兴奋，本章我们不打算构建自己完整成熟的查询提供器。不过，在理解本节的所有内容后，如果你需要的话完全有能力去构建一个——可能更重要的是，在你发出LINQ to SQL查询的时候，能理解将发生的事情。在执行的时候，查询提供器最艰巨的工作就是需要解析表达式树并把它们转换为用于目标平台的适当形式。我们现在来关注一些在那之前发生的事情——LINQ执行查询之前的准备工作。

我们将编写IQueryable和IQueryProvider的实现，并尝试对它们运行几个查询。有趣的地方不是结果——在执行这些查询的时候，我们什么都没有做，而是构成查询过程的一系列调用。我们会编写FakeQueryProvider和FakeQuery类型。每个接口方法的实现都把当前涉及的表达式用简单的日志记录方法打印出来（这里未显示）。

首先让我们看一下FakeQuery，如代码清单12-3所示。

代码清单12-3　记录方法调用的IQueryable的简单实现

```
class FakeQuery<T> : IQueryable<T>
{
   public Expression Expression { get; private set; }        ◁┐
   public IQueryProvider Provider { get; private set; }        │
   public Type ElementType { get; private set; }               ❶ 声明简单的自动属性
```

① 是那些能保持延迟执行的操作符的用途，比如Where和Join。稍后，我们将会看到在诸如Count这样的聚合操作。

② Execute和CreateQuery都具有泛型和非泛型的重载。使用非泛型版本，在代码中动态地创建查询会更容易。编译时查询表达式使用泛型版本。

```
    internal FakeQuery(IQueryProvider provider,
                       Expression expression)
    {
       Expression = expression;
       Provider = provider;
       ElementType = typeof(T);
    }

    internal FakeQuery() : this(new FakeQueryProvider(), null)
    {
       Expression = Expression.Constant(this);        ❷ 使用这个查询作为初始表达式
    }

    public IEnumerator<T> GetEnumerator()
    {
       Logger.Log(this, Expression);
       return Enumerable.Empty<T>().GetEnumerator();
    }
                                                       ❸ 返回空结果序列
    IEnumerator IEnumerable.GetEnumerator()
    {
       Logger.Log(this, Expression);
       return Enumerable.Empty<T>().GetEnumerator();
    }

    public override string ToString()                ❹ 为了进行日志记录而进行覆盖
    {
       return "FakeQuery";
    }
 }
```

IQueryable的属性成员在FakeQuery中通过自动属性来实现❶，并通过构造函数来设置值。有两个构造函数：无参数的构造函数，主程序用它为查询创建普通“数据源”；而FakeQueryProvider则会调用另一个构造函数并传入当前的查询表达式。

Expression.Constant(this)用作初始数据源表达式❷只是为了展示查询表示原始的对象。(例如，假设某个实现表示一个数据表——如果不使用任何查询操作符，查询将返回整个数据表。)在常量表达式被记录的时候，将使用覆盖后的ToString❹。因此，我们要给出一个简短且固定的描述。这让最终的表达式比不使用重载方法时更加清晰。在我们被要求遍历查询结果的时候，简便起见，我们总是返回空序列❸。生产代码应该在这里解析表达式，或（很可能）调用查询提供器上的Execute方法，并返回结果。

可以看到，在FakeQuery中没有太复杂的东西，而代码清单12-4显示了同样简单的FakeQueryProvider。

代码清单12-4　使用FakeQuery来实现IQueryProvider

```
class FakeQueryProvider : IQueryProvider
{
   public IQueryable<T> CreateQuery<T>(Expression expression)
   {
      Logger.Log(this, expression);
      return new FakeQuery<T>(this, expression);
```

```
    }

    public IQueryable CreateQuery(Expression expression)
    {
        Type queryType = typeof(FakeQuery<>).MakeGenericType(expression.Type);
        object[] constructorArgs = new object[] { this, expression };
        return (IQueryable)Activator.CreateInstance(queryType, constructorArgs);
    }

    public T Execute<T>(Expression expression)
    {
        Logger.Log(this, expression);
        return default(T);
    }

    public object Execute(Expression expression)
    {
        Logger.Log(this, expression);
        return null;
    }
}
```

对于FakeQueryProvider的实现可谈的东西甚至比FakeQuery还少。CreateQuery方法不执行真正的处理，而是作为查询的工厂方法。唯一棘手的地方在于非泛型重载仍然需要为FakeQuery<T>（基于给定表达式的Type属性）提供正确的类型参数。Execute重载方法只是在记录调用日志后返回空结果。通常情况下，在这里应该完成大量的分析工作，以及对Web服务、数据库或任何目标平台的实际调用。

尽管我们没有完成实际的工作，当开始在查询表达式中把FakeQuery用作数据源的时候，有趣的事情就发生了。我无意中说过，我们如何在不显式地编写方法的情况下，通过写查询表达式来处理标准查询操作符：一切都是靠扩展方法，这次则是使用Queryable类包含的扩展方法。

12.2.3　把表达式粘合在一起：**Queryable**的扩展方法

正如Enumerable类型包含着关于IEnumerable<T>的扩展方法来实现LINQ标准查询操作符一样，Queryable类型包含着关于IQueryable<T>的扩展方法。IEnumerable<T>和Queryable的实现之间有两个巨大的区别。

首先，Enumerable的方法都使用委托作为参数，例如，Select方法使用Func<TSource, TResult>。这对于内存中的操纵是没有问题的，但对于在别处执行查询的LINQ提供器来说，我们需要能执行更详细检查的格式——表达式树。例如，Queryable中相应的Select的重载就要获取类型为Expression<Func<TSource,TResult>>的参数。编译器根本不会关心这些——在查询转译之后，它具有一个需要作为参数传递给方法的Lambda表达式，而Lambda表达式既可以被转换为委托实例，也可以转换为表达式树。

这就是LINQ to SQL能够如此无缝地工作的原因。涉及的4个关键元素都是C# 3的新特性：Lambda表达式、将查询表达式转换为使用Lambda表达式的“普通”表达式、扩展方法和表达式树。不能同时具有这4个元素，就会出现问题。例如，如果查询表达式总是被转化为委托，它们

就不能用于LINQ to SQL这样的提供器，因为这些提供器需要表达式树。图12-3显示了查询表达式选择的两种路径，它们唯一的区别就是数据源实现的接口不同。

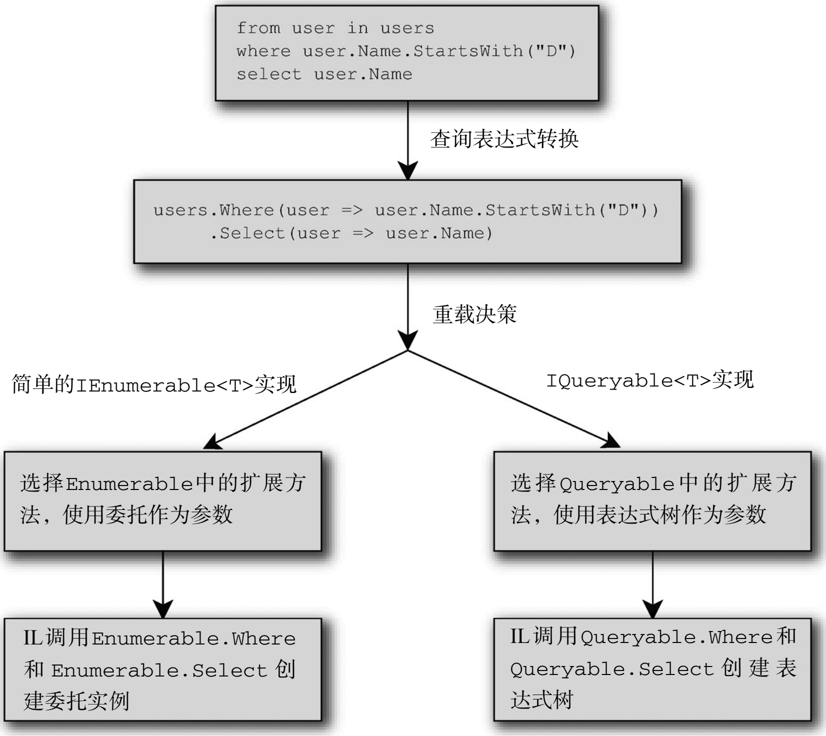


图12-3　选择两种路径的查询，具体选择哪个路径取决于数据源实现的是IQueryable接口还是只是IEnumerable

注意，图12-3中编译过程的前面部分是独立于数据源的。使用的查询表达式相同，转译方法也完全一致。只有在编译器查看转换后的查询，以查找适当的Select和Where方法来使用时，数据源才起到了重要的作用。此时，Lambda表达式既能被转换为委托实例也能被转换为表达式树，这样就有可能得到完全不同的实现：左边路径用于执行内存中的操作，而右边路径用来在数据库上执行SQL。

图12-3这里只是为了加深印象，实际上是使用Enumerable还是Queryable在C#编译器中并没有明显的支持。它们并不是仅有的两条路径，稍后我们还会看到并行LINQ和响应式LINQ。你也可以遵循这种查询模式创建自己的接口，实现扩展方法，或者创建包含适当实例方法的类型。

Enumerable和Queryable之间的第2个重大差别是，Enumerable的扩展方法会完成与对应查询操作符相关的实际工作（至少会构建完成这些工作的迭代器）。例如，Enumerable.Where中的代码执行特定的过滤操作，并在结果序列中生成适当的元素。通过比较，Queryable中的查询操作符的“实现”做的事情非常少：正如在12.2.1节结尾处所述的那样，它们仅仅基于参数创

建一个新的查询，或在查询提供器上调用Execute。换句话说，它们只用来构建查询和要执行的请求——不包含操作符背后的逻辑。这意味着，它们可用于任何使用表达式树的LINQ提供器，但是它们单独使用时没有任何意义。它们是代码和提供器细节之间的黏合剂。

有了Queryable扩展方法，以及我们的IQueryable和IQueryProvider实现，终于可以看看在用我们自定义的提供器构建查询表达式时，会发生什么事了。

12.2.4　模拟实际运行的查询提供器

代码清单12-5显示了一个简单查询表达式，用于（假设要）在我们的模拟数据源中查找以“abc”开头的所有字符串，并把结果投影到包含匹配字符串的长度数值的序列中。我们遍历这个结果，不过不用它们做任何事情，这是因为已经知道它们是空的。当然，我们没有源数据，并且也没有编写任何代码来执行真正的过滤——我们只是记录在创建查询表达式并遍历结果的过程中，LINQ所进行的调用。

代码清单12-5　使用模拟查询类的简单查询表达式

```
var query = from x in new FakeQuery<string>()
            where x.StartsWith("abc")
            select x.Length;
foreach (int i in query) { }
```

你期望代码清单12-5的运行结果是什么？特别是，你希望最后记录的应该是什么？我们通常会希望在什么地方用表达式树做一些真正的工作？下面是代码清单12-5的输出结果，为了能说得清楚明白，我们重新整理了一下格式：

```
FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))

FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)

FakeQuery<Int32>.GetEnumerator
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)
```

要记住的两件重要事情是，GetEnumerator只在最后才调用，而不在任何中间查询中调用，并且在GetEnumerator被调用的时候，我们已经有了出现在原始查询表达式中的所有信息。这样，就不必在每一步中手动地记录表达式的前面部分——到目前为止，单一的表达式树已经捕获了所有的信息。

顺便说一下，不要被这个简洁的输出结果所迷惑——实际的表达式树可能非常深且复杂，特别是在Where子句包含额外方法调用的时候。LINQ to SQL检查表达式树以算出应该执行什么样的查询。当调用CreateQuery时，LINQ提供器能够构建它们自己的查询（以它们需要的任何形式），不过，当调用GetEnumerator的时候，看一下最后的表达式树，可知道它们通常都比较简单，这是因为所有需要的信息都已经保存在同一个地方了。

代码清单12-5中记录的最后调用是FakeQuery.GetEnumerator，你也许会奇怪为什么需要

关于IQueryProvider的Execute方法。原因是这样的，并不是所有的查询表达式都生成序列——如果你使用Sum、Count或Average这样的聚合操作符，就不再真正创建一个“数据源”——我们会立刻对结果进行计算。这个时候Execute就会被调用，如代码清单12-6及其结果所示。

代码清单12-6　IQueryProvider.Execute

```
var query = from x in new FakeQuery<string>()
            where x.StartsWith("abc")
            select x.Length;
double mean = query.Average();

// Output
FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))

FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)

FakeQueryProvider.Execute
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
                    .Select(x => x.Length)
                    .Average()
```

如果要理解C#编译器针对查询表达式在幕后做了哪些工作，FakeQueryProvider是非常有用的。它显示了在查询表达式中引入的透明标识符，以及SelectMany、GroupJoin这样的经过转换的调用。

12.2.5　包装IQueryable

我们没有编写任何真正的查询提供器用于完成有用工作所需要的大量代码，不过希望我们的模拟提供器能让你知晓，LINQ提供器如何从查询表达式中获得信息。这一切都是通过Queryable扩展方法来构建的，并给出IQueryable和IQueryProvider适当的实现。

与本章剩余部分相比，本节的内容将更加详细，因为涉及我们在前面看到的LINQ to SQL代码的基础知识。尽管你基本不需要自己去实现查询接口，但执行C#查询表达式所涉及的步骤和（在执行时）在数据库上运行这些SQL的意义重大，是C# 3的这些重要特性的核心内容。理解C#为什么会获得这些特性，有助于你更好使用这门语言。

至此，我们对于LINQ如何使用表达式树的学习就告一段落了。本章剩余部分将介绍使用委托的进程内查询。你会看到，对于如何使用LINQ，仍然可以有很多变化和创新。第一站是LINQ to XML，它“仅仅”是用来与LINQ to Objects集成的XML API。

12.3　LINQ 友好的 API 和 LINQ to XML

LINQ to XML是目前我用过的最舒服的XML API。不管是消费已有的XML，抑或生成新的文档，还是两者兼而有之，它都易用且易懂。其中一部分原因与LINQ完全无关，但大部分原因还是因为它能与LINQ完美地交互。与12.1节一样，我将仅介绍一些足够理解示例的信息，然后看看

LINQ to XML如何将其自身的查询操作符与LINQ to Objects的查询操作符相混合。学习完本节之后，对于如何让自己的API与框架融合贯通，你一定会有自己的认识。

12.3.1　LINQ to XML中的核心类型

LINQ to XML位于System.Xml.Linq程序集，并且大多数类型都位于`System.Xml.Linq`命名空间[①]。该命名空间下几乎所有类型都以`X`为前缀；普通DOM API中的`XmlElement`类型在LINQ to XML中对应的是`XElement`。这样，即便你还没有立即熟悉确切的类型，也能很容易辨认出代码正在使用的是LINQ to XML。图12-4展示了最常用的一些类型。

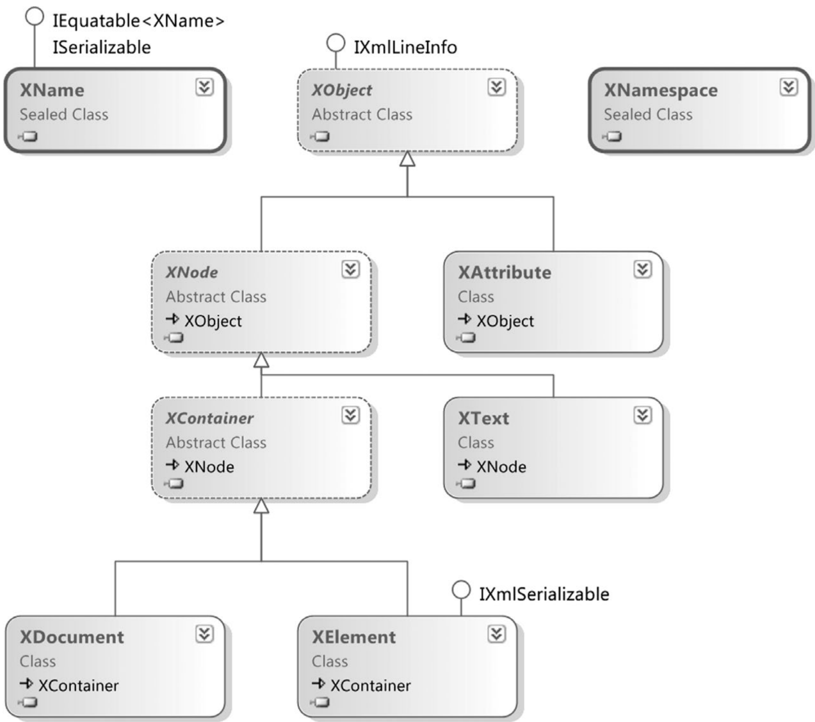


图12-4　LINQ to XML的类图，展示了最常用的类型

以下是对这些类型的概述。

- ❑ `XName`表示元素和特性的名称。创建实例时，通常使用字符串的隐式转换（这时不需要使用命名空间）或重载的`+(XNamespace, string)`操作符[②]。
- ❑ `XNamespace`表示XML命名空间，通常是一个URI。创建其实例时常常使用字符串的隐式

① 我经常忘了到底是`System.Xml.Linq`还是`System.Linq.Xml`。我想说的是，如果你知道它首先是一个XML API，就应该能记住其命名空间——不过这对我来说显然没用，希望你的运气比我好。

② 该操作符重载是属于`XNamespace`的。——译者注

转换。

- XObject是XNode和XAttribute的共同父类：与在DOM API中不同，在LINQ to XML中特性不是节点。如果某方法返回子节点的元素，这里面是不包含特性的。
- XNode表示XML树中的节点。它定义了各种用于操作和查询树的成员。XNode还有很多子类没有在图12-4中列出，如XComment和XDeclaration。它们相对来说并不常用，文档、元素和文本才是最常用的节点类型。
- XAttribute表示包含名/值对的特性。值从本质上来说是文本，但可以显式转换成其他数据类型，如int和DateTime。
- XContainer是XML树中可以包含子内容（主要为元素或文档）的节点。
- XText表示文本节点，其派生类XCData表示CDATA文本节点。（CDATA节点大致相当于逐字的字符串字面量，不需要任何转义。）XText很少直接在用户代码中实例化，当将字符串用于元素或文档的内容时，会将其转换为XText实例。
- XElement表示元素。它和XAttribute是LINQ to XML中最常用的类。与在DOM API中不同，在创建一个XElement时，不需要创建包含它的文档。如果你不是确实需要一个文档对象（如自定义XML声明），只用元素就可以了。
- XDocument表示文档。可以通过Root属性访问其根元素，相当于XmlDocument.DocumentElement。如前所述，你并不总是需要创建一个文档。

以上这些是最重要的类型，还有很多类型甚至可用于文档模型，另外还有一些类型可用于加载和保存选项。上面这些类型之中，你常常需要显式引用的只有XElement和XAttribute。若使用命名空间的话，还会用到XNamespace，而其余时间大多数其他类型都可以忽略。用寥寥几个类型就可以实现如此多的功能，真是令人惊叹啊！

说到惊叹，我不得不向你介绍LINQ to XML对命名空间的支持。我们并不打算到处使用命名空间，但它说明了如果转换和操作符设计良好，可以大大简化操作。它还使我们轻松进入第一个话题：构建元素。

如果在指定元素或特性名称时不需要指定命名空间，就可以只用字符串表示。然而你找不到任何一个类型的构造函数包含string参数——它们接收的都是XName。我们可以将string隐式转换为XName或XNamespace。将命名空间与字符串相加仍然可以得到一个XName。在操作符滥用和妙用之间存在一条微妙的分界线，而LINQ to XML将这一点体现得尤为明显。

下面的代码创建了两个元素，一个使用了命名空间，一个没有使用：

```
XElement noNamespace = new XElement("no-namespace");
XNamespace ns = "http://csharpindepth.com/sample/namespace";
XElement withNamespace = new XElement(ns + "in-namespace");
```

即使使用了命名空间，以上的代码也十分易读，这比使用其他API要轻松得多。现在我们只创建了两个空元素，如何向它们添加内容呢？

12.3.2 声明式构造

在DOM API中，我们通常创建一个元素，然后向其中添加内容。在LINQ to XML中我们也可

以这样做，使用继承自XContainer的Add方法，但这并不是LINQ to XML的惯用法[①]。不过还是有必要看一下XContainer.Add的签名，因为它使用了内容模型。你也许会认为其签名为Add(XNode)或Add(XObject)，但事实上它只是Add(object)。XElement（和XDocument）的构造函数签名也使用了同样的模式。在名称之后，你可以什么都不指定（创建空元素），也可以指定一个对象（创建包含单个子节点的元素），或对象数组（创建包含多个子节点的元素）。在创建多个子节点的时候，使用了参数数组（C#中的params关键字），这意味着编译器将为我们创建数组，我们只需要不断列出参数即可。

使用简单的object作为内容类型，这听上去有点疯狂，但却极为有用。在创建内容时，不管是通过构造函数还是Add方法，都要考虑以下几点。

- 空引用会被忽略。
- XNode和XAttribute实例可以直接添加。如果它们已经有了父元素，将会被复制，但除此之外不需要任何转换。（编译器会执行一些完整性检查，如确保不会在一个元素中出现重复的特性。）
- 字符串、数字、日期、时间等将使用标准XML格式转换为XText。
- 如果参数实现了IEnumerable（并且没有被其他东西所覆盖），Add方法将迭代其内容，并添加各个值，必要的时候会使用递归。
- 其他没有特殊处理的对象将调用ToString()将其转换为文本。

这意味着你没有必要在将内容添加到元素之前进行特殊的准备——LINQ to XML会自动为我们做出正确的处理。文档中已经明确地写出了细节，因此不必担心它过于神秘，不过它确实很神奇。

构造内嵌的元素可以使代码很自然地形成树形的层次结构。事实胜于雄辩，以下是LINQ to XML代码片段：

```
new XElement("root",
   new XElement("child",
      new XElement("grandchild", "text")),
   new XElement("other-child"));
```

下面是所创建的XML元素，注意代码和结果，它们看上去是如此相似：

```
<root>
  <child>
    <grandchild>text</grandchild>
  </child>
  <other-child />
</root>
```

到目前为止一切都很好，但对我们来说最重要的是上面列表中的第四项，即何时会递归地处理序列。因为它可以让你很自然地通过LINQ查询来构建XML结构。例如，本书网站上包含一些代码，可以从数据库中生成RSS源。构建XML文档的语句有28行之多——通常我认为这会让人深

① 很遗憾XElement没有实现IEnumerable，否则还可以使用集合初始化器来构造元素。不过没关系，使用构造函数已经可以很巧妙地做到了。

恶痛绝，但此处读起来却十分愉快[①]。该语句包含两个LINQ查询，一个用来生成特性值，另一个提供元素的序列，每个元素代表一个新项。你所看到的代码，与最终的XML十分相似。

为了更具体一些，我们举两个缺陷跟踪系统中的列子。我将使用LINQ to Objects示例数据进行演示，你也可以对其他LINQ提供器使用几乎完全相同的查询。我们首先构建一个元素，包含系统中的所有用户。在本例中，我们只需要一个投影，因此下面的代码清单12-7使用了点标记。

代码清单12-7　从示例用户中创建元素

```
var users = new XElement("users",
    SampleData.AllUsers.Select(user => new XElement("user",
       new XAttribute("name", user.Name),
       new XAttribute("type", user.UserType)))
);
Console.WriteLine(users);

// 输出
<users>
  <user name="Tim Trotter" type="Tester" />
  <user name="Tara Tutu" type="Tester" />
  <user name="Deborah Denton" type="Developer" />
  <user name="Darren Dahlia" type="Developer" />
  <user name="Mary Malcop" type="Manager" />
  <user name="Colin Carton" type="Customer" />
</users>
```

如果想创建复杂一点儿的查询，可以使用查询表达式。代码清单12-8创建了另一个用户列表，这次只包含SkeetSoft的开发者。为了有所区别，这次每个开发者的名字都是元素内的文本节点，而不再是特性值。

代码清单12-8　创建文本节点元素

```
var developers = new XElement("developers",
   from user in SampleData.AllUsers
   where user.UserType == UserType.Developer
   select new XElement("developer", user.Name)
);
Console.WriteLine(developers);

// 输出
<developers>
  <developer>Deborah Denton</developer>
  <developer>Darren Dahlia</developer>
</developers>
```

类似的操作可以应用于所有示例数据，文档结构如下所示：

12

① 确保可读性好的一个因素是，我创建了一个扩展方法，将匿名类型转换为元素，将其属性作为子元素。这段代码位于MiscUtil项目中（参见http://mng.bz/xDMt），如果你感兴趣，可以随时查看。它只有在XML结构满足某种特定模式时才有帮助，可以显著地减少杂乱的`XElement`构造函数调用。

```
<defect-system>
  <projects>
    <project name="..." id="...">
      <subscription email="..." />
    </project>
  </projects>
  <users>
    <user name="..." id="..." type="..." />
  </users>
  <defects>
    <defect id="..." summary="..." created="..." project="..."
            assigned-to="..." created-by="..." status="..."
            severity="..." last-modified="..." />
  </defects>
</defect-system>
```

在下载的解决方案的XmlSampleData.cs中包含生成所有数据的代码。它没有使用“一条长语句”的方法，而是分别创建顶级元素下的各个元素，然后再将它们粘合在一起，如下所示：

```
XElement root = new XElement("defect-system", projects, users, defects);
```

我们将使用这段XML来演示下一个与LINQ的结合点：查询。先从单个节点的查询方法开始。

12.3.3　查询单个节点

你大概会期望XElement实现IEnumerable，这样就可以免费使用LINQ查询了。事实可没有那么简单，因为对于XElement来说，可迭代的东西太多了。XElement包含很多轴方法（axis method），可用于查询资源。如果你熟悉XPath，应该对轴的概念不陌生。

以下是可以直接对单个节点执行查询的轴方法，每个方法都返回适当的IEnumerable<T>。

- Ancestors
- DescendantNodes
- Annotations
- Elements
- Descendants
- ElementsBeforeSelf
- AncestorsAndSelf
- DescendantNodesAndSelf
- Attributes
- ElementAfterSelf
- DescendantsAndSelf
- Nodes

这些轴方法的功能不言自明（详细内容可查阅MSDN文档）。它们包含很多有用的重载，可以仅获取拥有适当名称的节点，如对XElement调用Descendants("user")将返回该元素下的所有user元素。

除了这些返回序列的方法，有些方法还返回单个结果——其中最重要的是Attribute和Element，分别返回已命名的特性和具备指定名称的第一个子元素。此外，还存在从XAttribute或XElement到int、string、DateTime等类型的显式转换。这对结果的过滤和投影来说十分重要。在转换到可空值类型时还包含到等价的空类型的转换——如果对空引用调用转换，将返回空值（转换为字符串时也是如此）。这种空的传递性意味着你不必在查询中检查特性或元素是存在还是不存在，只需要使用查询的结果代替即可。

这与LINQ有什么关系呢？多个查询结果以IEnumerable<T>返回，意味着可以在查询出一些元素之后，使用普通的LINQ to Objects方法。代码清单12-9展示了如何查找用户名称和类型，这次使用了XML中的示例数据。

代码清单12-9　显示XML结构中的用户

```
XElement root = XmlSampleData.GetElement();

var query = root.Element("users").Elements().Select(user => new
            {
              Name = (string) user.Attribute("name"),
              UserType = (string) user.Attribute("type")
            });
foreach (var user in query)
{
   Console.WriteLine ("{0}: {1}", user.Name, user.UserType);
}
```

在创建了数据之后，我们向下导航到users元素，并获取它的直接子元素。这两步可以缩短为root.Descendants("user")，不过了解更严格的导航也是很有益处的，可以在必要的时候使用。并且这样写也更健壮，以防止文档结构发生变化，如在文档中其他位置添加了另一个（不相关的）user元素。

查询表达式的其余部分只是XElement到匿名类型的投影。我得承认在处理用户类型的时候要了一点小手段：我将其保存为字符串，而没有调用Enum.Parse将其转换为适当的UserType值。后一种方法完全没有问题，只是在仅需要字符串形式的时候略显啰嗦，很难在打印页严格的限制之内进行合理的格式化。

这没有什么特别之处，毕竟将查询结果作为序列返回是很常见的。但要注意特定领域查询操作符是如何与通用操作符无缝集成的。这并不是故事的结尾，LINQ to XML还包含一些额外的扩展方法。

12

12.3.4　合并查询操作符

我们看到，查询的部分结果往往为另一个序列，而在LINQ to XML中则通常为元素的序列。如果想要对所有这些元素执行XML指定的查询该如何操作呢？举个略显虚构的例子，可以使用root.Element("project").Elements()找出示例数据中的所有项目，但如何找出各个项目中的subscription元素呢？这时我们需要对各个元素执行另一个查询，然后合并这些结果。

（同样，我们也可以使用root.Descendants ("subscription")，但对于更复杂的文档模型，它可能无法工作。）

这听起来似乎很熟悉，确实，LINQ to Objects已经提供了SelectMany操作符（在查询表达式中使用多个from子句）来实现该功能。因此查询可以写为：

```
from project in root.Element("projects").Elements()
from subscription in project.Elements("subscription")
select subscription
```

由于项目下除了订阅信息外没有其他元素，因此可以使用不指定子元素名称的Elements重载。我认为在本例中指定名称会更清晰，不过这只是个人习惯而已。当然，在调用Element("projects").Elements("project")的时候也可以指定同一个参数名。以下是同一个查询，使用点标记和仅返回合并序列的SelectMany重载，没有执行进一步的投影：

```
root.Element("projects").Elements()
    .SelectMany(project => project.Elements("subscription"));
```

无论如何，这两种查询并非完全没有可读性，但它们均不理想。LINQ to XML提供了一些扩展方法（位于System.Xml.Linq.Extensions类中），有的针对特殊的序列类型，有的是包含强制类型参数的泛型方法，以应对C# 4之前缺乏泛型接口协变性的问题。这其中包含一个InDocumentOrder方法，顾名思义，可以按照文档中元素的顺序排序。12.3.3节中提到的轴方法大多可作为扩展方法的形式使用。这意味着我们可以将查询转换为下面这种简单形式：

```
root.Element("projects").Elements().Elements("subscription")
```

这种构造使你可以很容易地在LINQ to XML中编写XPath风格的查询，无须所有类型都为字符串。如果要使用XPath，也可以通过更多的扩展方法实现——我发现查询方法大多数时候让我受益匪浅。它还支持与LINQ to Objects的查询操作符相结合。例如，要找到所有项目名中含“Media”的订阅信息，可以这样：

```
root.Element("projects").Elements()
    .Where(project => ((string) project.Attribute("name"))
                                   .Contains("Media"))
    .Elements("subscription")
```

在介绍并行LINQ之前，我们先来思考LINQ to XML如何设计才配得上LINQ这个名号——并且考虑我们如何才能将同样的技术应用于自己的API中。

12.3.5　与LINQ和谐共处

如果将LINQ to XML孤立地看成某XML API的一部分，可能会觉得某些设计很奇怪，但在LINQ的大背景下，这些设计都恰如其分。设计者无疑设想了如何在LINQ查询中使用这些类型，以及它们如何与其他数据源交互。如果你需要编写自己的数据访问API，不管出于什么样的上下文，也都应该考虑同样的事宜。如果有人在查询表达式中使用了你的方法，它们是否返回了有用的东西？在流畅的表达式中，它们是否可以使用你编写的查询方法，再使用LINQ to Objects中的方法，然后再使用你编写的方法？

LINQ to XML使用了如下三种方式与其他LINQ相适应。

- 在构造函数中消费序列。LINQ是刻意声明式语言，LINQ to XML支持声明式地创建XML结构。
- 在查询方法中返回序列。这大概是数据访问API必须遵循的最为明显的步骤：查询结果应该轻而易举地返回`IEumerable<T>`或实现了该接口的类。
- 扩展了可以对XML类型的序列所作的查询，这样可以让它们看上去更像是统一的查询API，尽管有些查询必须用于XML。

你也许会思考其他可以让你的库在LINQ中运转良好的方式：以上这些并不是唯一的选择，却是很好的入门方法。重要的是，在我的劝导下，你开始设身处地地站在这样一位开发者的角度思考问题：渴望在使用了LINQ的代码中使用自己的API。这样一位开发者想达到什么样的目标呢？你的API可以和LINQ简单地融合吗？或者它们根本就是风马牛不相及？

我们这辆介绍LINQ不同方法的旋风大巴目前已经行驶了将近一半的旅程了。下一站是纠结站，它既让你心安理得，又让你惴惴不安：我们又回到了对简单序列的查询上，但这一次，是并行的……

12.4　用并行 LINQ 代替 LINQ to Objects

我跟进并行LINQ（Parallel LINQ）已经有一段时间了。我是在Joe Duffy于2006年9月发表的一篇博文（参见http://mng.bz/vYCO）中第一次听说这项技术的。2007年11月，并行LINQ发布了第一个社区技术预览版（CTP，Community Technology Preview），截至目前，所有的特性都经历了较长时间的演变。它现在是更广阔的并行扩展（Parallel Extension）的一部分，后者属于.NET 4，旨在为并行编程提供构建块，比迄今为止一直在使用的相对较小的基元集要更为高级。并行扩展比并行LINQ（常称作PLINQ）的范畴更广泛，不过我们在此只关注其LINQ部分。

并行LINQ的背后理念是，某个LINQ to Objects查询需要执行很长的时间，而使用多线程利用多核优势进行查询则可以运行得很快，并且改动也很少。和其他与并发有关的技术一样，并行LINQ并不是很简单，但你一定会对它所实现的一切感到惊讶。当然，我们仍然在冥思苦想比独立LINQ更强大的技术——我们在考虑不同的交互模型，而不再是精确的细节。如果你对并发有兴趣，我由衷地建议你研究并行扩展——它是最近我遇到的最有前途的并行方案。

我将在本节中使用一个单独的示例：呈现曼德博罗特集的图像（参见http://en.wikipedia.org/wiki/Mandelbrot_set）。在向复杂的领域迈进之前，我们先在单线程中正确实现它。

12.4.1　在单线程中绘制曼德博罗特集

在数学家抨击我之前，我要先声明一点：在此使用的是不严谨的曼德博罗特集。以下是具体细节，但这并不是十分重要：

- 我们要创建一个矩形图像，给出各种选项，如宽度、高度、源和搜索深度；

- 对于图像中的每个像素，都将计算一个字节值，代表256色调色板中的某个索引；
- 一个像素值的计算不依赖于其他结果。

最后一点至关重要——它意味着该任务是完全并行的。换句话说，在任务内部，没有什么可以阻止它并行执行。我们还需要一种机制，用来跨线程分配工作量，然后将结果收集在一起，但其余的都应该很简单。这种机制就是PLINQ，它负责分发和校对，我们只需要表示要处理的部分。

为了演示多种方法，我将一个抽象基类（负责设置任务、运行查询和显示结果），以及一个用于计算单个像素颜色的方法组合在一起。还有一个抽象方法负责创建包含值的字节数组，可以转换为图像。首先是第一行像素，从左至右，接着是第二行，以此类推。本节的每个示例都只是实现了该方法而已。

需要指出的是，这里使用LINQ并不是一个理想的解决方案——会有很多低效的情况出现。不要把注意力放到这些方面，要集中于这个想法上，即我们要做完全并行的查询，并且要跨多核来执行。

代码清单12-10从各个方面显示了单线程的版本。

代码清单12-10　单线程的曼德博罗特生成查询

```
var query = from row in Enumerable.Range(0, Height)
            from column in Enumerable.Range(0, Width)
            select ComputeIndex(row, column);

return query.ToArray();
```

我们遍历每一行以及每行中的每一列，计算相关像素的索引。调用`ToArray()`计算结果序列，并将其转换为数组。图12-5显示了所生成的美妙结果。

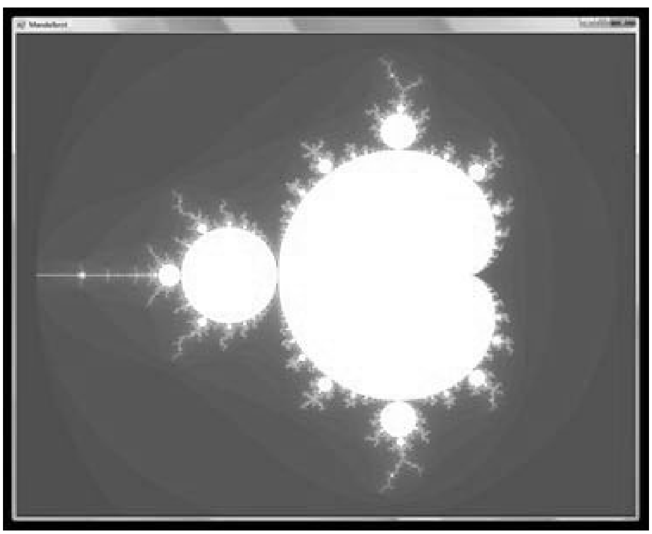


图12-5　单线程生成的曼德博罗特集图像

我的双核笔记本电脑生成这幅图用了5.5秒；为了使时间差异更为明显[1]，ComputeIndex方法执行了一些不必要的迭代。现在我们拥有了时间和结果的基准，下面尝试将查询并行化。

12.4.2 ParallelEnumerable、ParallelQuery和AsParallel

并行LINQ带来了一些新的类型，但大多数情况下，你都不会看到它们的名称。它们位于System.Linq命名空间，因此你甚至不需要更改using指令。ParallelEnumerable是一个静态类，与Enumerable类似。它里面几乎全部是扩展方法，其中大多数都扩展了ParallelQuery这个类型。

该类型包含泛型和非泛型形式（ParallelQuery<TSource>和ParallelQuery），我们大多数情况下都会使用其泛型形式，就像IEnumerable<T>要比IEnumerable常用。此外，还有一个OrderedParallelQuery<TSource>类，它是IOrderedEnumerable<T>的并行版本。这些类型之间的关系如图12-6所示。

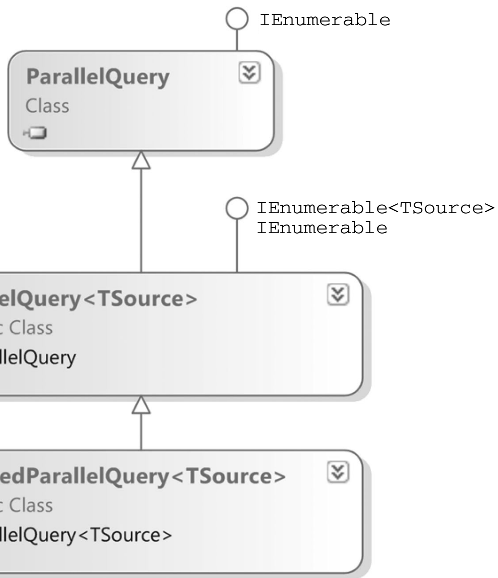


图12-6 并行LINQ的类图及其与普通LINQ接口的关系

如图所示，ParallelQuery<TSource>实现了IEnumerable<TSource>，因此如果你恰当地构建了一个查询，就可以用普通的方式对结果进行迭代。如果是并行查询，ParallelEnumerable中的扩展方法要优先于Enumerable中的（因为ParallelQuery<T>比IEnumerable<T>更特殊；如果你忘了规则，可以参考10.2.3节）——这就是并行机制维护整个查询的原理。所有的LINQ

① 得到正确的基准是很难的——尤其是牵扯到线程的情况下。我并没有做严格的测量或类似的工作。给出的时间只是为了反映快慢，请谨慎地对待这些数字。

标准查询操作符都有并行的版本，然而如果创建了自己的扩展方法，你应该格外小心。你仍然可以调用它们，但从这一刻开始，查询将被强制为单线程的。

那么如何以并行查询开始呢？答案是调用AsParallel，它是ParallelEnumerable中的扩展方法，扩展了IEnumerable<T>。因此我们可以异常简单地将曼德博罗特查询并行化，如代码清单12-11所示。

代码清单12-11　第一次尝试多线程的曼德博罗特生成查询

```
var query = from row in Enumerable.Range(0, Height)
                                   .AsParallel()
            from column in Enumerable.Range(0, Width)
            select ComputeIndex(row, column);

return query.ToArray();
```

搞定了吗？好像还没有。该查询确实可以并行运行——但结果并不完全符合我们的要求：它的顺序与我们处理每行的顺序并不相同。我们没能得到美妙的曼德博罗特图像，而是得到了如图12-7所示的东西，并且每次的结果都不相同。

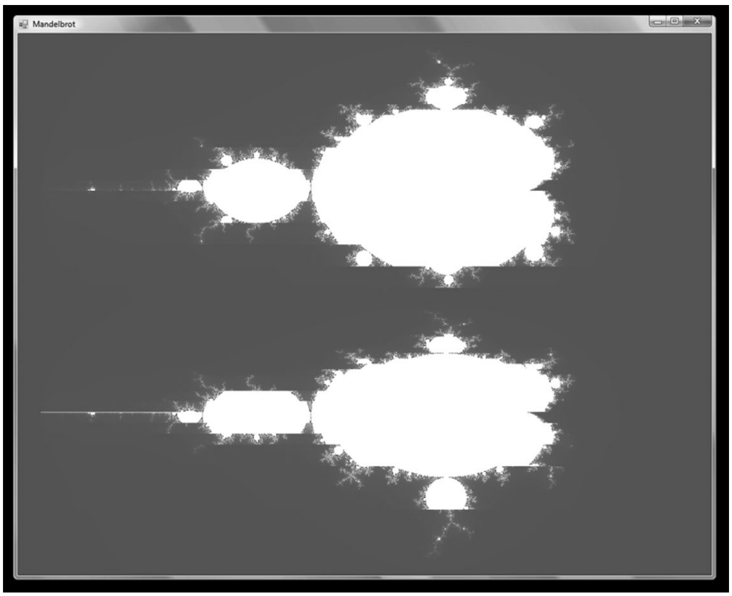


图12-7　使用无序查询生成的曼德博罗特图像，结果是某些部分位置错位

从好的方面看，这次呈现共耗时3.2秒，我的电脑显然使用了两个核。但另一方面，得到正确的结果才是更重要的。

这是并行LINQ斟酌再三之后的特性，听到这里你想必会瞠目结舌。对并行查询排序需要在线程之间进行更多的调和，而并行的唯一目标就是为了改善性能，因此PLINQ默认使用无序查询。不过这在本例中就有点讨厌了。

12.4.3 调整并行查询

幸运的是，可以避开这一点——你只需要使用AsOrdered扩展方法，强制对查询排序即可。代码清单12-12展示了修改后的代码，它可以产生原始的图像。它比无序查询要略慢，但仍明显快于单线程版本。

代码清单12-12 有序的多线程曼德博罗特查询

```
var query = from row in Enumerable.Range(0, Height)
                                   .AsParallel().AsOrdered()
            from column in Enumerable.Range(0, Width)
            select ComputeIndex(row, column);

return query.ToArray();
```

排序的细微差别超出了本书的介绍范围，但我建议你读一下这篇博文（http://mng.bz/9x9U），它深入剖析了细节。

另外，还有很多方法可以改变查询的行为。

- AsUnordered——使有序查询变得无序；如果你只需要对查询的第一部分排序，该方法可以使后续部分更加高效。
- WithCancellation——在该查询中指定取消标记（cancellation token）。取消标记的使用贯穿了整个并行扩展，使任务以安全、可控的方式得以取消。
- WithDegreeOfParallelism——指定执行查询的最大并发任务数。如果你不想让你的机器疲于招架，或者对没有CPU限制的查询提高使用的线程数量，这可以用来限制使用的线程数。
- WithExecutionMode——强制查询按并行方式执行，即使并行LINQ认为单线程执行得更快。
- WithMergeOptions——可以改变对结果的缓冲方式：禁止缓冲可以尽量缩短第一条结果的返回时间，但却降低了总的效率；完全缓冲的效率最高，但在查询执行完毕之前，不会返回任何结果。默认情况下使用两者的折中方案。

重要的是，除了排序，这些不应该影响到查询的结果。你可以设计LINQ to Objects中的查询和测试，然后并行化，按要求排序，并视情况调整。如果你将最终的查询展示给了解LINQ但不懂PLINQ的人看，你只需要解释PLINQ相关的方法调用——他们对查询的其他部分很熟悉。你见过这么简单就实现并发的吗？（并行扩展其他部分的目的，也是在可能的情况下实现简单化。）

说明 **与代码共舞** 可下载的源代码中演示了几个更深层次的情况：如果查询并行化是针对像素的，而不是针对行，无序查询会显得更加怪异；与Enumerable.Range(…).AsParallel()相比，ParallelEnumerable.Range方法可以为PLINQ提供更多的信息。我在本节使用的是AsParallel()，因为它是较常见的将查询并行化的方式——大多数查询都不是从范围开始的。

将进程内（in-process）查询模型从单线程转换到并行并没有多少概念上的跳跃，然而在下一节，我们会将模型掉一个个儿。

12.5　使用 LINQ to Rx 反转查询模型

目前我们看到的所有LINQ库都有一个共同点：所得到的数据为`IEnumerable<T>`。乍看上去，这似乎显而易见、不值一提——还会有其他选择吗？不过，如果我们是推数据，而不是拉数据呢？与数据消费者掌管一切不同，数据提供者处于主导地位，当新数据可用的时候，由数据消费者进行响应（react）。这听上去与之前完全不同，不过不必担心：你实际上接触过它的基本概念——以事件形式。如果你对事件的订阅、响应和取消订阅轻车熟路，实际上你已经有了一个很好的起点。

.NET 的 Reactive Extension 是微软的一个 DevLabs 项目（参见 http://mng.bz/R7ip 和 http://mng.bz/HCLP），包含用于.NET 3.5 SP1、.NET 4、Silverlight 3和4，甚至面向JavaScript的版本。现在，获取最新版本的最简单方法就是访问NuGet。你可能听说过它的很多名称，最常见的缩写是Rx和LINQ to Rx，这也是我这里要用的名称。它的适用范围不仅仅局限于这里提到的响应端，尤其是还有一个有趣的程序集叫做System.Interactive，它包含各种额外的LINQ to Objects方法；System.Reactive实现了各种推操作。我们对推模型的介绍将只是点到为止。我清楚本章的所有概念都是点到为止，但对于本节来说，这个词儿尤其适用：库本身就有很多东西需要学习，不仅如此，它还是一种完全不同的思维方式。Channel 9上有大量的视频（参见http://channel9.msdn.com/tags/Rx/）——有一些基于数学理论，不过，也有一些更贴近实际。本节我将重点强调将LINQ概念应用于数据流推模型的方式。

介绍得已经够多了，下面我们来看看构成LINQ to Rx基础的两个接口。

12.5.1　`IObservable<T>`和`IObserver<T>`

LINQ to Rx的数据模型与普通`IEnumerable<T>`的模型在数学上是对偶的（mathematical dual）[①]。在开始对拉集合进行迭代时，我们以“请给我一个迭代器”（调用GetEnumerator）开始，然后重复“还有其他项吗？如果有，就给我”（调用MoveNext和Current）。LINQ to Rx则是反向的。它不向迭代器发出请求，而是提供一个观察者。然后，它也不请求下一个项，而是通知你的代码是否准备好了一个项、是否有错误发生、是否到达了数据末端。

以下是涉及的两个接口的声明：

```
public interface IObservable<T>
{
   IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<T>
{
```

① 要对这种对偶性和LINQ自身的本质有更多的了解，我推荐Bart de Smet的博客文章“The Essence of LINQ—MINLINQ”，网址是http://mng.bz/96Wh。

```
    void OnNext(T value);
    void OnCompleted();
    void OnException(Exception error);
}
```

这两个接口属于.NET 4（位于System命名空间），但LINQ to Rx的其余部分则是需要单独下载的。实际上，在.NET 4中这两个接口是IObservable<out T>和IObserver<in T>，分别表示IObservable是协变的，IObserver是逆变的。下一章中我们将了解更多关于泛型可变性的内容，而这里为了简便起见，假设它们是不变的。

图12-8通过各自模型中的数据流展示了这种对偶性。

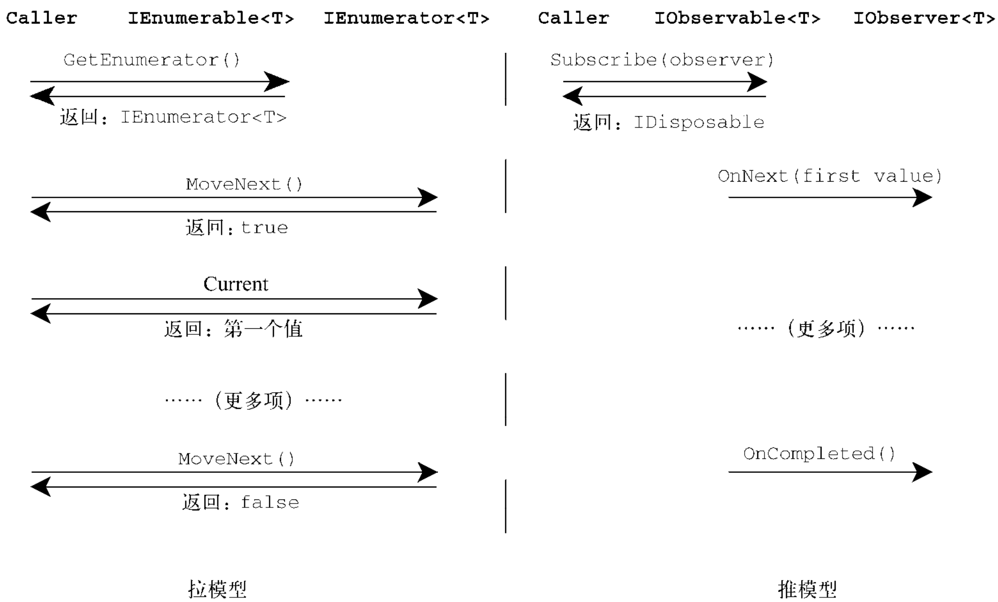


图12-8 用序列图展示IEnumerable<T>和IObservable<T>的对偶性

我想肯定不是只有我一个人认为推模型难以理解，因为它天生具备异步处理的能力，但从流程图中可以看出，它比拉模型要简单得多。部分原因是由于拉模型中的方法太多：如果IEnumerator<T>中存在签名为bool TryGetNext(out T item)的方法，将会简单一些。

前面提到过，LINQ to Rx与我们熟悉的事件十分类似。调用一个可观察对象（observable）的Subscribe，就像是对事件使用+=来注册处理程序一样。Subscribe返回的可处置（disposable）值会记住传入的观察者（observer）：处置它就像对同一个处理程序使用-=一样。在很多情况下，你都不需要取消对可观察对象的订阅；只有在对某个序列处理到一半时取消订阅，即相当于提前跳出foreach循环时，它才真的有用。对你来说，如果处置某个IDisposable值失败，可能是个大麻烦，但在LINQ to Rx中通常是安全的。本章中的示例都不会使用Subscribe的返回值。

对于IObservable<T>来说就是这些内容，但观察者本身又如何呢？它为什么有三个方法呢？在普通的拉模型中，调用MoveNext/Current时可能发生如下三种情况：

- 位于序列末尾，这时MoveNext返回false；
- 未到达序列末尾，这时MoveNext返回true，Current返回新的值；
- 出现错误——可能由于网络连接等问题，导致读取下一行失败，这时将抛出异常。

IObserver<T>接口分别用不同的方法来代表这几种情况。通常，观察者将重复调用OnNext方法，并最终调用OnCompleted——这期间如果出现了某种错误，就用OnError代替。在序列结束或发生错误之后不会再调用其他方法。不过，你基本不需要直接实现IObserver<T>。IObservable<T>包含很多扩展方法，其中包括对Subscribe的重载[①]。我们可以通过提供适当的委托，来订阅可观察对象：通常你需要给出一个可供所有项执行的委托，然后再提供一个在完成时或错误时执行，或同时用于两种情况的委托，后者是可选的。

在介绍了一些不同寻常的理论之后，我们来看一些LINQ to Rx的实际代码。

12.5.2 简单的开始

我们打算用与LINQ to Objects相同的方式来演示LINQ to Rx——使用范围。只不过使用Observable.Range来创建一个可观察的范围，而不再使用Enumerable.Range。每当一个观察者订阅这个范围时，使用OnNext把数字发送给该观察者，最后将调用OnCompleted。我们会尽可能保持简单，仅仅在接收到值的时候进行打印，并在最后或发生错误时打印确认信息。

它所用的代码确实比拉模型要短，代码清单12-13展示了这一点。

代码清单12-13 初次接触IObservable<T>

```
var observable = Observable.Range(0, 10);
observable.Subscribe(x => Console.WriteLine("Received {0}", x),
                     e => Console.WriteLine("Error: {0}", e),
                     () => Console.WriteLine("Finished"));
```

在本例中，我们不容易产生错误，不过完整起见，我还是保留了错误通知的委托。结果如下：

```
Received 0
Received 1
...
Received 9
Finished
```

Range方法返回的是一个冷可观察对象（cold observable）。它处于休眠状态，直到某个观察者订阅了它，它才会向该观察者发送值。如果其他观察者也订阅了该对象，将会得到该范围的一个副本。这与点击按钮这种普通的事件不太相同，对于后者，多个观察者可以同时订阅同一个实际的值序列——并且即便没有任何观察者，也会有效地产生值。（毕竟，就算没有附加任何事件处理程序，你也可以点击按钮。）这种序列称为热可观察对象（hot observable）。知道你正在处理的是哪种类型是很重要的，即使对这两种类型应用了相同的操作。

现在我们完成了最简单的事情，接下来尝试一些我们熟悉的LINQ操作符。

① IObservable<T>的扩展方法位于System.Reactive.dll程序集中的System.Linq.Observable和System.ObservableExtensions类中。——译者注

12.5.3 查询可观察对象

你肯定已经熟悉了这种模式，即在一个静态类（Observable，估计你可以猜到）中编写各种扩展方法，执行适当的转换。我们仅介绍一些可用的操作符，然后考虑哪些不可用，并探究其原因。

1. 过滤和投影

我们直接使用接收一个数字序列的查询表达式，过滤掉奇数，并对剩下的数字求平方。我们用Console.WriteLine订阅最终的查询结果，这样所生成的每一项都能显示出来。代码清单12-14展示了具体的代码，注意这个查询表达式与LINQ to Objects是何其相似。

代码清单12-14 在LINQ to Rx中使用过滤和投影

```
var numbers = Observable.Range(0, 10);
var query = from number in numbers
            where number % 2 == 0
            select number * number;
query.Subscribe(Console.WriteLine);
```

简便起见，我没有为完成和错误添加处理程序，为了保持代码优雅简洁，还使用了从Console.WriteLine方法组到Action<int>的转换。所产生的结果与LINQ to Objects相同：0,4,16……。下面我们来看看分组。

2. 分组

LINQ to Rx中的group by查询表达式将为每个组生成新的IGroupedObservable<T>，尽管接下来对分组的处理并不总是很明显。例如，使用嵌套的订阅可以在新分组产生的时候向其订阅一个观察者，这是很常见的。每个分组构造接收并产生结果——做出某种重定向选择，就像剧场的引座员检查每个人的票，并把他们带到剧场的相关位置。相比之下，LINQ to Objects在返回之前把整个分组收集在一起，这意味着要对结果进行缓冲，直到序列的末尾。

代码清单12-15展示了一个嵌套订阅的示例，同时还演示了如何生成分组结果。

代码清单12-15 对3取模并分组

```
var numbers = Observable.Range(0, 10);
var query = from number in numbers
            group number by number % 3;
query.Subscribe(group => group.Subscribe
    (x => Console.WriteLine("Value: {0}; Group: {1}", x, group.Key)));
```

我们在LINQ to Objects中处理分组时常常要嵌套foreach循环，因此在LINQ to Rx中要嵌套订阅，这样对比可能会有助于你理解上面的代码。

如果有疑问，可以试着找出这两种数据模型之间的对偶性。在LINQ to Objects中，我们通常依次处理每个分组，而用LINQ to Rx则顺序显示，代码清单12-15的结果将如下所示：

```
Value: 0; Group: 0
Value: 1; Group: 1
Value: 2; Group: 2
Value: 3; Group: 0
```

```
Value: 4; Group: 1
Value: 5; Group: 2
Value: 6; Group: 0
Value: 7; Group: 1
Value: 8; Group: 2
Value: 9; Group: 0
```

这对理解推模型有着重要的意义，并且在某些情况下，在LINQ to Objects中所需的大量的数据缓冲操作，都可以用LINQ to Rx更高效地实现。

作为最后的示例，我们来看另一个使用了多序列的操作符。

3. 合并

LINQ to Rx提供了SelectMany的一些重载，其理念仍然与LINQ to Objects相同：原始序列中的每一项都生成一个新的序列，最终的结果是所有这些新序列的组合。下面的代码清单展示了这一点——它有点类似代码清单11-16，那是我们第一次介绍LINQ to Objects中的SelectMany。

代码清单12-16　用SelectMany生成多个范围

```
var query = from x in Observable.Range(1, 3)
            from y in Observable.Range(1, x)
            select new { x, y };
query.Subscribe(Console.WriteLine);
```

以下是输出结果，你应该能够猜到：

```
{ x = 1, y = 1 }
{ x = 2, y = 1 }
{ x = 2, y = 2 }
{ x = 3, y = 1 }
{ x = 3, y = 2 }
{ x = 3, y = 3 }
```

在本例中，结果是确定的，但这仅仅是由于默认情况下，Observable.Range在当前线程生成各个项。你完全有可能在多个线程环境中产生多个序列。

例如，你可以修改对Observable.Range的第二次调用，指定Scheduler.ThreadPool作为第三个参数。这时，尽管每个内部序列按自身顺序出现，但不同的序列之间彼此会混合在一起。假设在一个体育场里，某个裁判陆续为多个不同的竞赛项目叩响发令枪：尽管你知道每项比赛的胜利者，但却不知道哪项比赛会率先结束。

这些也许会让你晕头转向。不用紧张，其实我也一样，不过同时我还被它深深地吸引住了。

4. 新引入的和不支持的

我们都知道let子句只能在调用Select时使用，但LINQ to Rx并没有实现所有的LINQ to Objects操作符。漏掉的大多是那些缓冲输出结果并返回新的可观察对象的操作符。例如Reverse方法和OrderBy方法。C#对此表示无压力——你只是不能再对基于可观察对象的查询表达式使用orderby子句了。LINQ to Rx包含Join方法，但它并不直接处理可观察对象，而是处理连接计划（join plan）。这是Rx实现连接演算（join-calculus）的部分内容，超出了本书的范畴。此外，Rx也没有实现GroupJoin方法，因此也不支持join...into。

要了解查询表达式语法不包含的LINQ标准查询操作符，以及它所支持的其他方法，可以查

看`System.Reactive`文档。尽管你开始时可能会因为LINQ to Rx不支持很多在LINQ to Objects中熟悉的功能（通常是由于它们在Rx中没有什么意义）而感到失望，但你马上会惊诧于它提供了如此丰富的可用方法。许多新方法都被移植到了LINQ to Objects的`System.Interactive`程序集中。

12.5.4　意义何在

我很清楚目前还没有提供任何令人信服的理由来让你使用LINQ to Rx。我是有意为之，因为我不打算展示一个完整有用的示例——这部分内容只是本章的一个附带产品，不应该占用太多篇幅。但Rx提供了一种优雅的方式来思考各种异步处理——如普通.NET事件（可以使用`Observable.FromEvent`将其视为可观察对象）、异步I/O和调用Web服务。它提供了一种有效的方式来管理复杂性和并发。毫无疑问，它比LINQ to Objects要难以理解，但如果你所处的环境正好可以使用Rx，则说明你正面对的是高度复杂的情形。

LINQ to Rx是一个发展历程相对较短的项目，它第一次发布在DevLabs上是2009年11月。如果对这个简短的介绍感兴趣，那你一定要仔细研究一番。我之所以在本书中涵盖Rx，并不是为了面面俱到，而是因为它展示了LINQ为什么会设计成这种方式。尽管`IEnumerable<T>`和`IObservable<T>`之间存在相互转换的方法，但它们没有继承关系。如果编程语言要求LINQ所涉及的类型必须为拉序列，那么根本不会诞生任何Rx查询表达式。如果扩展方法在某种程度上仅局限于`IEnumerable<T>`，后果就更加不堪设想。此外我们还看到，并不是所有LINQ操作符都可用于Rx——因此，支持对给定的提供器有意义的内容，语言以这种模式来指定查询转译，是十分重要的。我希望你意识到尽管推模型和拉模型“势不两立”，但LINQ在某种程度上扮演了“统一原力”（unifying force）[①]的作用。

最后一个话题要简单得多——我们又回到了LINQ to Objects，但这一次我们要自己编写扩展方法。

12.6　扩展 LINQ to Objects

LINQ最美妙的一点就是其可扩展性。你不仅可以编写自己的查询提供器和数据模型，还可以向已有的提供器和模型中添加新的内容。根据我的经验，这通常用于LINQ to Objects。如果你要查询不直接支持的特殊类型（或用标准查询操作符不合适、效率偏低），可以自己进行扩展。当然，编写通用的泛型方法要比仅解决当前的问题更具挑战性，但如果你发现重复编写了相似的代码，就有必要考虑将其重构为新的操作符了。

我个人很喜欢编写查询操作符。它很有技术挑战性，却不需要大量的代码——而且结果还可以很优雅。我们将在本节介绍一些能够使你的自定义操作符行为高效且可预知的方式，然后给出一个从序列中选择随机元素的完整示例。

① unifying force一词来源于詹姆斯·卢西诺的星战小说，参见http://en.wikipedia.org/wiki/The_Unifying_Force。——译者注

12.6.1　设计和实现指南

本节大部分内容似乎都是显而易见的，不过，我们在此总结了编写操作符时应注意的问题列表，这相当有用。

1. 单元测试

为操作符编写一套优秀的单元测试通常是很简单的，尽管原本简单的代码最终的单元测试数量可能会相当惊人。不要忘记测试个别情况，如空序列和无效参数等。MoreLINQ在其单元测试项目中包含了一些辅助方法，可用于我们自己的测试。

2. 检查参数

好的方法会检查传入的参数。但这对LINQ操作符来说有一个问题。我们已经看到，很多操作符都返回一个序列，而实现这种功能最简单的方式就是迭代器块。但你应该在调用方法的同时执行参数检查，而不应该等到调用者决定迭代其结果的时候。如果打算使用迭代器块，就把方法分成两部分：在公共方法中执行参数检查，然后调用一个私有方法进行迭代。

3. 优化

`IEnumerable<T>`本身所支持的操作十分有限，但你所操作的序列的执行时类型可能具备更多的功能。例如，`Count()`操作符总是可用的，但通常其复杂度为`O(n)`。然而如果调用的是`ICollection<T>`实现，就可以直接使用其`Count`属性，复杂度为`O(1)`。在.NET 4中，这种优化也包括`ICollection`。同样，通过索引获取特定的元素一般会很慢，但如果序列实现为`IList<T>`，就会很高效。

如果操作符能从这些优化中受益，你就可以根据不同的执行时类型来选择不同的执行路径。假如要在单元测试中测试缓慢的路径，也可以调用`List<T>`的`Select(x=>x)`来得到一个非列表序列。`LinkedList<T>`可以测试实现了`ICollection<T>`而没有实现`IList<T>`的情况。

4. 文档

在文档中指明代码对输入的处理和操作符的预期性能是十分重要的。如果你的方法需要处理多个序列，这样做就尤其重要：对哪个序列先求值？进行多长时间？代码对数据是进行流处理、缓冲，还是两者兼而有之？是延迟执行还是立即执行？参数可以为空吗？如果可以，有什么特殊含义吗？

5. 尽量只迭代一次

可以对`IEnumerable<T>`进行多次迭代——实际上对于同一个序列，你可以同时拥有多个活动的迭代器。但对于一个查询操作符来说，这样做可不是什么好主意。如果可能的话，对输入序列只迭代一次是很明智的选择。这意味着你的代码甚至可以用于不可重复的序列，如从网络流中读取多行。如果你确实需要多次读取序列（并且不想像`Reverse`那样缓冲整个序列），最好在文档中特别注明这一点。

6. 释放迭代器

在大多数情况下，我们可以使用`foreach`语句来迭代数据源。但有时我们需要对第一个元素进行不同的处理，这时直接使用迭代器可以使代码更加简单。在这种情况下，要为迭代器使用

using块。我们不习惯自行释放（dispose）迭代器，因为通常foreach为我们做了这些，但这样却很难发现某些bug。

7. 自定义比较器

很多LINQ操作符都包含可以指定适当IEqualityComparer<T>或IComparer<T>的重载。如果你是在为别人（可能是你无法接触的开发者）构建通用的库，提供类似的重载是很有价值的。另一方面，如果你是唯一的用户，或者代码即将成为团队的一部分，这也是需要实现的。不过它很简单：通常简单的重载只需要调用复杂的重载，传入EqualityComparer<T>.Default或Comparer<T>.Default作为比较器。

纸上得来终觉浅，绝知此事要躬行。下面让我们来看一个简单的示例。

12.6.2　示例扩展：选择随机元素

我们的扩展方法思路十分简单：接收一个序列和一个Random实例，返回序列中的一个随机元素。你可以添加不需要Random实例的重载，但我更倾向于显式地依赖随机数生成器。由于多种原因随机性是一个复杂的话题，我不会在此讨论，你可以参考本书网站上的文章（参见http://mng.bz/h483）。篇幅所限，代码清单12-17中没有包含XML文档和单元测试，你可以在下载的代码中找到它们。

代码清单12-17　从序列中选择随机元素的扩展方法

```
public static T RandomElement<T>(this IEnumerable<T> source,
                                 Random random)
{
   if (source == null)                                    <-❶ 验证参数
   {
      throw new ArgumentNullException("source");
   }
   if (random == null)
   {
      throw new ArgumentNullException("random");
   }
   ICollection collection = source as ICollection;        <-❷ 优化集合
   if (collection != null)
   {
      int count = collection.Count;
      if (count == 0)
      {
         throw new InvalidOperationException("Sequence was empty.");
      }
      int index = random.Next(count);                     用ElementAt
      return source.ElementAt(index);                     <- 进一步优化
   }
   using (IEnumerator<T> iterator = source.GetEnumerator())   <-❸ 处理低效的情况
   {
      if (!iterator.MoveNext())
      {
         throw new InvalidOperationException("Sequence was empty.");
```

12

```
        }
        int countSoFar = 1;
        T current = iterator.Current;
        while (iterator.MoveNext())
        {
            countSoFar++;
            if (random.Next(countSoFar) == 0)      ❹ 有时需要取代当前的推测
            {
                current = iterator.Current;
            }
        }
        return current;
    }
}
```

代码清单12-17并没有将扩展方法划分为参数验证和实现两部分，因为它没有使用迭代器块。关于这一点可以回过头去看看10.3.3节Where操作符的实现。这里也没有使用自定义比较器，除此之外，均符合我们上一节所列出的注意事项。

我们首先显式地校验参数❶。在第15章，我们将学习另一种表示前置条件的方式——代码契约，不过在此我们还是使用普通的异常。有趣的地方在❷——我们处理了原序列实现ICollection的情况[①]。这样我们就可以很容易进行计数，然后生成一个随机数来决定取出哪个元素。我们没有显式地处理原序列实现IList的情况，而是使用ElementAt来进行处理（就像注释中提到的那样）。

如果我们处理的序列不是集合（如其他操作符的结果），我们想避免先计数再取元素，这就需要缓冲序列内容，或者迭代两次。相反，我们只用一次，显式地获取迭代器❸，这样就可以简单地测试空序列的情况。该方法的绝妙之处[②]是❹，我们有1/*n*的概率会用当前迭代器中的元素替换已选出的随机元素，*n*是已经迭代的元素数量。因此有1/2的概率会用第二个元素替换第一个，有1/3的概率用第三个元素替换前两个元素的结果，以此类推。最终的结果是，序列中每个元素被选择的机会是均等的，而且我们只迭代了一次。

当然重点并不在于这个方法做了什么，而在于我们实现的时候所思考的潜在问题。当你明白了这些问题之后，实现这样一个健壮的方法就不费吹灰之力了。久而久之，你个人的工具库就会越来越充实。

12.7　小结

本章包含的内容和本书剩余部分有很大的不同。我们没有详细深入某个单一主题，而是以浅显的方式讲述了大量的LINQ技术。

我并不期望你对这里看到的任何一个特定的技术都特别熟悉，不过还是希望你深刻理解LINQ为何重要。它不是一种关于XML、内存数据查询、SQL数据查询、可观察对象，甚至是枚

① 下载代码中还包含了实现ICollection<T>的同样的测试，就像.NET 4中的Count()操作符一样。代码块完全相同，只是使用了不同的类型和不同的变量名。

② 我可以说它绝妙，因为尽管这是我的实现，却不是我的创意。

举器的技术——它是关于表达式一致性的技术，并让C#编译器有机会至少在某种程度上去验证你的查询，而不用关心最终执行的平台。

你应该对表达式树有足够的重视，它们存在于与一些C#编译器直接紧密关联的框架元素中，如字符串、`IDisposable`、`IEnumerable<T>`和`Nullable<T>`。表达式树是实现LINQ行为的通行证，它让LINQ可以在本地机器上跨越功能边界，并表示由LINQ提供器所带来的某种外来语言中的逻辑。

不仅是表达式树——我们还依赖于由编译器提供的查询表达式转译，通过这个方式，Lambda表达式能被转译为委托或者表达式树。扩展方法也很重要，没有它们，每个提供器就必须给出所有相关方法的实现。如果回顾一下C#的新特性，你将发现它们几乎都以某种方式对LINQ做出了重大贡献。这也是本章存在的部分缘由：显示出C#所有新特性的连接关系。

其实，我本不该热情洋溢地赞扬LINQ这么久——它有一些优势，但也存在着一些缺点。LINQ不会一直支持我们表达在查询中所需的任何东西，它也不会隐藏底层数据源的所有细节信息。谈到数据库LINQ提供器的时候，过去导致开发人员陷入大麻烦的阻抗失配，现在还是困扰着我们：我们虽然能如此这般通过ORM之类的系统来降低它们的影响，但如果未能对这些查询有一个很好的理解，就可能遇到更严重的问题。尤其是，不要认为掌握了LINQ，你就不必再弄懂SQL——LINQ仅仅是在你对细节不感兴趣的时候，隐藏SQL的一种方式。同样，要实现有效的并行查询，你需要知道什么时候可以进行排序，什么时候不适宜排序，并通过一些调优信息来促进框架发展。

自从.NET 3.5发布以来，我很高兴看到社区已经全盘接受了它。C# 4有许多有趣的特性，在本书的最后一部分中，我们就来介绍它们。

Part 4

第四部分

C# 4：良好的交互性

C# 4是一个奇特的野兽。它既不像C# 2那样包含“多个、无关的、主要的新特性”，也不像C# 3那样“所有的特性都是为了LINQ”。相反，C# 4的新特性介于两者之间。互操作性是一大主题，但即便你从不需要与其他环境交互，很多特性也是十分有用的。

我最喜欢的C# 4特性是可选参数和命名实参。它们虽然相对简单，但在很多地方都可以很好地运用，改善代码的可读性，让你倍感身心愉悦。你是否还在费力地猜测每个实参的含义？给它们命名吧。你是否还在编写没完没了的重载，以避免调用者指定所有的参数？让一些参数成为可选的吧。

如果你要操作COM，那么C# 4将为你带来全新的气息。我刚才提到的这两个特性可以极大地简化某些API的使用，这些组件的设计者大多假设你使用的语言支持可选参数和命名实参。除此以外，部署也得到了改善，支持命名索引器，并且提供了一个方便的写法，以避免到处按引用传递参数。C# 4中最大的特性——动态类型——同样也可以简化与COM的集成。

我们将在第13章介绍这些领域，以及让人伤透脑筋的接口和委托的泛型可变性。不用担心，我会慢慢地讲解这些特性，而且最重要的是，大多数时候你都没有必要知道细节。它只是让你的代码得以实现在C# 3中不能实现的东西。

第14 章介绍动态类型和动态语言运行时（Dynamic Language Runtime，DLR）。这是一个庞大的话题。我将集中介绍C# 语言如何实现动态类型，不过也会看一些与IronPython 等动态语言交互的示例，以及类型如何动态地响应方法调用、属性访问等。这里有必要申明的是， 动态特性虽然很重要，但并不意味着你应该在代码中随处使用动态表达式。它不会像LINQ 那么普遍，但如果你确实需要动态类型，你会发现C# 4 提供了很好的实现。

第 13 章 简化代码的微小修改

本章内容

- ❑ 可选参数
- ❑ 命名实参
- ❑ 简化COM中的`ref`参数
- ❑ 内嵌的COM主互操作程序集
- ❑ 调用COM中声明的命名索引器
- ❑ 接口和委托的泛型变体
- ❑ 锁和字段风格的事件

和以前的版本一样，C# 4也包含一些小特性，它们并不值得独立成章。事实上，C# 4只有一个真正的大特性——动态类型，我们将在下一章进行介绍。这里要介绍的改动，只是使C#用起来更加舒服，特别是经常操作COM时。这些特性使代码更加清晰、避免了调用COM时的繁琐操作并且可以简化部署。

这些会让你兴奋得心跳加速吗？好像还不行。但它们确实是优秀的特性，并且其中一些将会得到广泛地应用。让我们先来看看如何调用方法。

13.1 可选参数和命名实参

可选参数（optional parameter）和命名实参（named argument）大概是C# 4特性里的蝙蝠侠和罗宾[①]。它们是不同的，但却经常成对出现。我先单独进行讨论，然后再在一些有趣的示例中将它们结合在一起。

① 或乡村骑士和丑角，如果你觉得这样可以显得更有修养。（罗宾是蝙蝠侠的助手，他们联手化解了很多危机；独幕歌剧《乡村骑士》和《丑角》的演出时间都不长，而且都是以当代为背景的写实歌剧，因此经常被视为二联剧在舞台上演出。作者在这里用这两个比喻来说明可选参数和命名实参通常要配合使用。——译者注

参数和实参

本节显然会多次讨论参数和实参。在非正式场合，这两个术语通常可以互换，但这里我会使用它们的正式定义。参数（也称为形式参数）变量是方法或索引器声明的一部分，而实参是调用方法或索引器时使用的表达式。例如下面的代码片段：

```
void Foo(int x, int y)
{
    // 使用x和y执行一些操作
}
...
int a = 10;
Foo(a, 20);
```

这里的参数是x和y，实参是a和20。

我们先来看看可选参数。

13.1.1　可选参数

Visual Basic早就支持可选参数，CLR也从.NET 1.0开始就有了这个特性。顾名思义，可选参数意味着一些参数是可选的，调用者不必显式地指定它们的值。对于任何这种参数，都将给定一个默认值。

1. 动机

如果某个操作需要多个值，而有些值在每次调用的时候又往往是相同的，这时通常可以使用可选参数。假设我们要读取一个文本文件，你可能会提供一个方法，允许调用者指定文件的名称和编码方式。但编码通常为UTF-8，如果能在需要的时候自动使用这种编码方式，一切就很完美了。

以前在C#中实现这种功能通常使用的是方法重载：声明一个包含所有可能参数的方法，其他方法调用这个方法，并传递恰当的默认值。例如，你可能会创建这样的方法：

```
public IList<Customer> LoadCustomers(string filename,
                                     Encoding encoding)
{
    ...                                   <—— 在这里进行真正的处理
}

public IList<Customer> LoadCustomers(string filename)
{
    return LoadCustomers(filename, Encoding.UTF8);     <—— 默认编码方式为UTF-8
}
```

对于单个参数来说，这种方法没有问题，但如果数量过多，就显得有些棘手了。每个额外的参数都会使重载的数量翻倍；如果有两个参数的类型相同，还会出现问题，因为声明的这些方法将包含相同的签名。通常，同一组重载也需要多种参数类型。如`XmlReader.Create()`方法可以通过`Stream`、`TextReader`或字符串创建`XmlReader`，同时也提供了指定`XmlReaderSettings`和其他参数的重载。由于这种重复，该方法共包含12个重载。

可选参数可以显著地降低重载的数量。让我们来看看它是如何实现的。

2. 声明可选参数并在调用时省略它们

创建可选参数是非常简单的，只需使用类似变量初始化程序一样的语法提供一个默认值。图13-1展示了一个包含三个参数的方法，两个为可选，一个为必备。

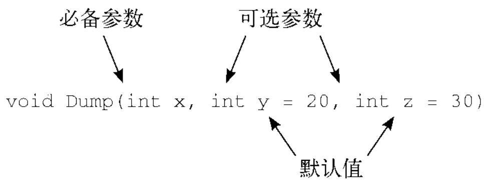


图13-1　声明可选参数

所有的方法都只是输出实参，但这已经足够了。代码清单13-1给出了完整的代码，并调用该方法3次，每次都指定了不同的实参数量。

代码清单13-1　声明包含可选参数的方法并调用

```
static void Dump(int x, int y = 20, int z = 30)
{
   Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
...
Dump(1, 2, 3);
Dump(1, 2);
Dump(1);
```

❶ 声明包含可选参数的方法
❷ 调用方法，给定所有的实参
❸ 省略一个实参
❹ 省略两个实参

指定了默认值的参数为可选参数❶。如果调用者没有指定y，它的初始值将为20，同样z的默认值为30。第一次调用❷显式指定了所有的实参，其余的调用（❸和❹）分别省略了1个和2个参数，因此将使用其默认值。当漏掉一个实参时，编译器假设省略的为最后一个参数，然后是倒数第二个，以此类推。因此输出结果为：

```
x=1 y=2 z=3
x=1 y=2 z=30
x=1 y=20 z=30
```

注意，尽管编译器可以对可选参数和实参进行一些智能分析，来确定省略的参数类型，但它并没有这么做：它假设我们提供的实参顺序与参数定义的顺序是一样的[①]。这意味着下面的代码是无效的：

```
static void TwoOptionalParameters(int x = 10,
                                  string y = "default")
{
   Console.WriteLine("x={0} y={1}", x, y);
}
...
TwoOptionalParameters("second parameter");    <── 错误
```

① 当然，除非使用命名实参，稍后我们将学习这点。

这里调用TwoOptionalParameters方法时为第一个实参指定了一个字符串，而该方法不包含第一个参数可转换为字符串的重载，因此将产生编译错误。这是一件好事——因为重载决策有些棘手（特别是包含泛型类型推断的时候），除非让编译器尝试所有不同的排列来找出可能调用的类型。如果你想省略某个可选参数的值而指定其后面的某个可选参数，则需要使用命名实参。

3. 可选参数的约束

可选参数包含一些规则。所有可选参数必须出现在必备参数之后，参数数组（用params修饰符声明）除外，但它们必须出现在参数列表的最后，在它们之前为可选参数。参数数组不能声明为可选的，如果调用者没有指定值，将使用空数组代替。可选参数还不能使用ref或out修饰符。

可选参数可以为任何类型，但对于指定的默认值却有一些限制。它们必须为常量：数字或字符串字面量、null、const成员、枚举成员和default(T)操作符。此外对于值类型，你还可以调用与default(...)操作符等价的无参构造函数。指定的值会隐式转换为参数类型，但这种转换不能是用户定义的。表13-1展示了一些有效的参数列表。

表13-1 使用可选参数的一些有效的方法参数列表

声　明	已构造类型的例子
`Foo(int x, int y = 10)`	默认值采用数字字面量
`Foo(decimal x = 10)`	从int到decimal的隐式内置转换
`Foo(string name = "default")`	默认值采用字符串字面量
`Foo(DateTime dt = new DateTime())`	DateTime的零值
`Foo(DateTime dt = default(DateTime))`	另一种零值语法
`Foo< T >(T value = default(T))`	类型参数的默认值操作符
`Foo(int? x = null)`	可空转换
`Foo(int x, int y = 10, params int[] z)`	在可选参数之后的参数数组

与此相反，表13-2展示了一些无效的参数列表以及相应的说明。

表13-2 使用可选参数的一些无效的方法参数列表

声　明	已构造类型的例子
`Foo(int x = 0, int y)`	非参数数组的必备参数出现在可选参数之后
`Foo(DateTime dt = DateTime.Now)`	默认值必须为常量
`Foo(XName name = "default")`	从string到XName的转换是用户定义的
`Foo(params string[] names = null)`	参数数组不能为可选的
`Foo(ref string name = "default")`	ref/out参数不能为可选的

默认值必须为常量会带来两种不同的问题，其中一个问题在不同的上下文中很常见，下面我们就来看看这个问题。

13

4. 版本化和可选参数

对可选参数默认值的约束非常类似对const字段和特性值的约束。在这两种情况下，当编译器引用这些值时，会直接将其复制到输出结果中。所生成的IL与源代码中包含默认值时是完全一致的。这意味着如果改变默认值而不重新编译引用它的代码，那么这些代码仍将使用旧的默认值。

具体来说，可遵循以下步骤。

(1) 创建包含类似下面这个类的类库（Library.dll）：

```
public class LibraryDemo
{
   public static void PrintValue(int value = 10)
   {
      System.Console.WriteLine(value);
   }
}
```

(2) 创建引用该类库的控制台应用程序（Application.exe）：

```
public class Program
{
   static void Main()
   {
      LibraryDemo.PrintValue();
   }
}
```

(3) 运行应用程序，如果不出意外将打印10。

(4) 像下面这样更改PrintValue的声明，然后只重新编译类库：

```
public static void PrintValue(int value = 20)
```

(5) 再次运行应用程序，仍然打印10。该值已经被直接编译到了可执行文件里。

(6) 重新编译应用程序，然后运行，这次将打印20。

这种版本化问题将导致难以跟踪的bug，因为所有的代码看上去都是正确的。基本上，你必须使用永远不会改变的真正常量作为可选参数的默认值[①]。这样做的好处是给了调用者一个保证：编译值即使用值。这对于开发者来说，比使用动态计算的值或依赖于执行时库版本的值要舒服得多。

当然，这也意味着你不能使用那些无法表示为常量的值——例如不能创建一个包含默认值为“当前时间”的方法。

5. 用可空性使默认值更加灵活

幸运的是，我们还可以突破默认值必须为常量的限制。我们引入一个魔值来表示默认值，然后在方法内部用真正的默认值替换这个魔值。如果魔值这个词给你带来了困扰，我一点儿也不会感到奇怪——这里我们使用null来作为魔值，因为它已经代表了正常值的缺失。如果参数类型为值类型，我们只需使用相应的可空值类型，这时仍然可以将默认值指定为null。

让我们来看看与引入整个话题时所使用示例的类似情况：允许调用者向某个方法提供一个适当的文本编码，默认值为UTF-8。然而我们不能用Encoding.UTF8来指定默认编码，因为它不是

① 或者在修改了默认值之后重新编译所有的程序集。在很多情况下，这也是一个合理的选择。

常量值。但可以使用一个空参数值，来表示“使用默认值”。为了演示如何处理值类型，可修改方法，使其对文本文件添加一个带时间戳的消息。编码的默认值为UTF-8，时间戳为当前时间。代码清单13-2显示了完整的代码及几个使用示例。

代码清单13-2 使用空默认值来处理非常量的情况

```
static void AppendTimestamp(string filename,
                            string message,
                            Encoding encoding = null,
                            DateTime? timestamp = null)
{
   Encoding realEncoding = encoding ?? Encoding.UTF8;
   DateTime realTimestamp = timestamp ?? DateTime.Now;
   using (TextWriter writer = new StreamWriter(filename,
                                               true,
                                               realEncoding))
   {
      writer.WriteLine("{0:s}: {1}", realTimestamp, message);
   }
}
...
AppendTimestamp("utf8.txt", "First message");
AppendTimestamp("ascii.txt", "ASCII", Encoding.ASCII);
AppendTimestamp("utf8.txt", "Message in the future", null,
                new DateTime(2030, 1, 1));
```

两个必备参数

❶ 两个可选参数

❷ 为了方便使用空合并操作符

❸ 显式使用null

代码清单13-2显示了这种方法的一些优秀特性。首先，解决了版本化的问题。可选参数的默认值为空❶，而有效值则为“UTF-8编码方式”和“当前日期和时间”。它们都不能表示为常量，在改变有效默认值时（如将本地时间改为当前UTC时间），就不再需要重新编译所有调用AppendTimestamp的程序集。当然，改变有效默认值会改变方法的行为，这与改变其他任何代码一样需要引起你的注意。这时，负责版本化的是你（库作者）——要确保不能破坏客户端。至少有一点你是知道的，所有调用者的行为都是一样的，不论是否重新编译。

代码中还引入了额外的灵活性。不仅可选参数能够缩短调用的代码，并且只要我们愿意就可以显式地指定“使用默认值”，来让方法选择适当的值。现阶段，如果不提供编码方式，则只能使用这种方式来显式指定时间戳❸，但命名实参将改变这一切。

使用空合并操作符❷可以简化对可选参数值的操作。为对打印结果进行格式化，本例使用了单独的变量，而在实际编码过程中，你完全可以在调用StreamWriter构造函数和WriteLine方法时直接使用相同的表达式。

这种方法有两个缺点：首先，如果调用者由于bug而不小心传入了null，将得到默认值而不是异常。对于可空值类型来说，调用者要么显式使用空，要么使用非空实参，这没什么大问题；但对于引用类型来说，问题就来了。

比如，你不想用null来表示“真正的”值[①]。有些时候你希望空即是空——如果不希望它作

① 我们差不多需要另一个类似null的特殊值，来表示“请使用该参数的默认值”，并允许该特殊值自动提供给缺失的实参或显式地指定在实参列表中。这肯定会带来很多问题，但却是一个有趣的想法。

13

为默认值，可以指定一个不同的常量[①]或者将参数设为必备的。而有些时候，你无法找到一个明显的可以一直作为正确默认值的常量，我建议你移除一些常见的困难，尽可能让可选参数保持简单。

我们稍后会研究一下可选参数是如何影响重载决策的，在此之前先来看看命名实参。

13.1.2　命名实参

命名实参的基本概念是，在指定实参的值时，可以同时指定相应参数的名称。编译器将判断参数的名称是否正确，并将指定的值赋给这个参数。这本身就可以在某些情况下提升可读性。实际上，命名实参常常与可选参数同时出现，但我们先来看看简单的情况。

说明　***索引器、可选参数和命名实参***　你可以在索引器和方法中使用可选参数和命名实参。但对于索引器来说，只有当参数多于一个时，这样做才有意义，因为在访问索引器时，不能一个参数也不指定。由于这种限制，我不认为这些特性会在索引器中广泛使用，因此本书不会对此进行介绍。但它仍然能够按你所期望的那样工作。

你肯定见过类似下面的代码：

```
MessageBox.Show("Please do not press this button again", // 文本
                "Ouch!"); // 标题
```

此处选择的示例是相当温和的。如果参数过多并且类型大都相同，则情况将更加糟糕。但即便只有这两个参数，在阅读代码时，仍然需要猜测各个参数的含义，除非像我那样进行注释。然而还有一个问题：注释不一定是正确的，没有什么能够验证它们。相反，命名实参却请出了编译器来帮助验证。

1. 语法

对于上面的示例，我们要做的仅仅是在各个实参之前加上它们的参数名称以及一个冒号，以使代码更为简洁：

```
MessageBox.Show(text: "Please do not press this button again",
                caption: "Ouch!");
```

不可否认，现在我们无法使用自认为最有意义的名称（比如我认为`title`比`caption`更合适），但至少不会再轻易出错了。

当然，最常犯的错误是把参数顺序弄反。如果没有命名参数，就会产生问题：消息框中的标题和内容是反的。但有了命名参数，参数顺序就是无关紧要的了。我们可以这样重写上面的代码：

```
MessageBox.Show(caption: "Ouch!",
                text: "Please do not press this button again");
```

正确的文本仍将出现在正确的位置，因为编译器会根据名称进行识别。

① 比如，用`int.MinValue`作为默认值。——译者注

再比如我们在代码清单13-2中调用的StreamWriter构造函数，其第二个实参为true——这是什么意思？强制在每次写入之后对流进行清理？包含字节顺序标记？还是将数据追加到现有文件，而不是新建文件？如果使用命名实参，可以像下面这样调用：

```
new StreamWriter(path: filename,
                 append: true,
                 encoding: realEncoding)
```

在这两个示例中，我们展示了命名实参如何将语义附加到值上。这在使代码能够更好地与人和计算机交互这条永无止境的道路上，迈出了坚实的一步。

当然，在参数含义明确时，没有必要使用命名实参。与所有特性一样，你都需要三思而后用。

包含out和ref的命名实参

如果要对包含ref或out的参数指定名称，需要将ref或out修饰符放在名称之后，实参之前。如对于int.TrayParse来说，代码如下：

```
int number;
bool success = int.TryParse("10", result: out number);
```

为了探索该语法的其他方面，代码清单13-3展示了一个包含三个整型参数的方法，与开始介绍可选参数的示例类似。

代码清单13-3　使用命名实参的简单示例

```
static void Dump(int x, int y, int z)          ❶ 正常声明方法
{
   Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
...
Dump(1, 2, 3);                                 ❷ 正常调用方法
Dump(x: 1, y: 2, z: 3);
Dump(z: 3, y: 2, x: 1);                        ❸ 为所有实参指定名称
Dump(1, y: 2, z: 3);
Dump(1, z: 3, y: 2);                           ❹ 为部分实参指定名称
```

每次调用代码清单13-3，所输出结果都是相同的：x=1，y=2，z=3。代码共使用了5种不同方式对相同的方法进行了有效地调用。值得注意的是，方法的声明没有任何特殊的地方❶：你可以对任何包含参数的方法使用命名实参。我们先用正常的方式调用方法，不使用任何特性❷。这是验证所有调用的结果是否全部相同的参照点。然后我们只使用命名实参对方法进行了两次调用❸。其中第二次打乱了实参的顺序，但得到的结果却是相同的，因为实参是按照参数的名称来匹配的，而不再是参数的位置。最后，我们混合使用命名实参和位置实参（positional argument），又进行了两次调用❹。未命名的实参称为位置实参，因此C# 3中所有有效的参数在C# 4看来都是位置实参。

图13-2展示了最后一行代码是如何工作的。

13

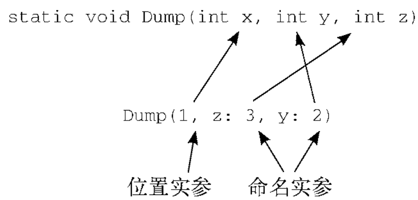


图13-2　同一个调用中的位置实参和命名实参

所有命名实参都必须位于位置实参之后，两者之间的位置不能改变。位置实参总是指向方法声明中相应的参数——你不能跳过参数之后，再通过命名相应位置的实参来指定。也就是说下面这两个调用都是无效的。

- `Dump(z: 3, 1, y: 2)`——位置实参必须在命名实参之前。
- `Dump(2, x: 1, z: 3)`——x已经由第一个位置实参指定了，因此不能再用命名实参指定。

现在，尽管在这种特殊情况下，方法的调用结果是相同的，但事实并非总是如此。下面我们来看看为什么重新排序的参数会改变行为。

2. 实参求值顺序

我们习惯于C#按实参的指定顺序对它们进行求值（evaluate），在C# 4之前，这与参数的声明顺序是一致的。而在C# 4中，只有前半句仍然成立：实参仍然按编写顺序求值，即使这个顺序有可能会不同于参数的声明顺序。如果实参求值产生了副作用，这一点就显得十分重要了。

应该避免实参的副作用，但有些情况下副作用却可以使代码变得清晰。也许更加实际的做法是，避免可能会互相干扰的副作用。但为了演示执行顺序，我们将打破这两个规则[①]，而在实际编码中则不应该这么做。

首先创建一个相对来说影响不大的示例，引入一个方法，记录输入内容并原封不动地返回——有点像复读机。我们将对该方法调用3次，并用3次的返回值调用`Dump`方法（该方法没有变化，因此不再展示）。代码清单13-4对`Dump`方法进行了两次调用，它们的输出结果略有不同。

代码清单13-4　记录实参求值

```
static int Log(int value)
{
   Console.WriteLine("Log: {0}", value);
   return value;
}
...
Dump(x: Log(1), y: Log(2), z: Log(3));
Dump(z: Log(3), x: Log(1), y: Log(2));
```

① 即避免副作用和避免互相干扰的副作用这两个规则。——译者注

代码清单13-4的输出结果展示了实际发生的事情：

```
Log: 1
Log: 2
Log: 3
x=1 y=2 z=3
Log: 3
Log: 1
Log: 2
x=1 y=2 z=3
```

在这两种情况下，`Dump`方法的参数x、y、z的值分别为1、2、3。然而我们可以看到，尽管在第一次调用时是按这种顺序求值的（与位置实参的顺序相同），但第二次调用最先求值的则是参数z。

我们可以使用一些可以改变实参求值结果的副作用，来使这种效果更为显著。如代码清单13-5所示，仍然使用同样的`Dump`方法。

代码清单13-5 滥用实参求值顺序

```
int i = 0;
Dump(x: ++i, y: ++i, z: ++i);
i = 0;
Dump(z: ++i, x: ++i, y: ++i);
```

代码清单13-5的结果可以用"惨不忍睹"来形容，就好像这段代码的维护者提着斧头追杀原作者后留下的血花四溅的凶杀现场。是的，从技术上来说最后一行会打印`x=2 y=3 z=1`，但你明白我所指的"结果"不是这个。我们要对这种代码说"不"。

为了增加可读性，我们必须想尽办法调整实参的顺序：比如在调用`MessageBox.Show`时，让标题出现在文本之前，这样才能更好地反映屏幕上的布局。但如果你的代码依赖于特殊的实参求值顺序，则应该引入一些局部变量，在单独的语句中执行相关的代码。编译器并不在乎是哪种方式，它会遵循规范的要求进行工作，不过，引入局部变量可以降低在重构时不经意间引入小bug的风险。

下面关注一点让大家兴奋的内容，我们将这两个特性（可选参数和命名实参）相结合，看看代码到底可以简洁到什么地步。

13.1.3 两者相结合

无须付出额外的努力就可以让可选参数和命名实参协同工作。我们经常遇到这种情况，很多参数都有明显的默认值，但我们却无法预测调用者会显式指定哪些参数。图13-3展示了几乎所有的组合：一个必备参数、两个可选参数、一个位置实参、一个命名实参以及一个缺失的使用可选参数默认值的实参。

在之前的代码清单13-2中，我们使用默认的UTF-8编码向每个文件追加一个时间戳。代码中使用`null`作为编码的实参，而此时此刻我们可以让代码更加简单，如代码清单13-6所示。

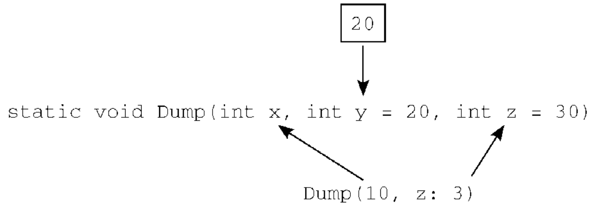


图13-3　命名实参和可选参数相结合

代码清单13-6　命名实参和可选参数相结合

```
static void AppendTimestamp(string filename,
                            string message,
                            Encoding encoding = null,
                            DateTime? timestamp = null)
{
                                                        跟之前的
                                                        实现一样
}
...
AppendTimestamp("utf8.txt", "Message in the future",    省略了编码方式
                timestamp: new DateTime(2030, 1, 1));   命名的时
                                                        间戳实参
```

这个示例过于简单，好处似乎不是那么明显，但如果想省略三四个实参而只指定最后一个，能使用命名实参简直就是一大幸事。

我们已经看到了如何使用可选参数减少重载数量，而不易变性也是值得一提的特定模式。

1. 不易变性和对象初始化

C# 4让我多少有些失望的地方是它没能简化不易变性（immutability）的实现。不易变类型是函数式编程的核心部分，而C#已经逐渐越来越多地支持函数化的风格。对象和集合初始化程序简化了易变（mutable）类型的使用，但不易变类型仍然被遗弃在阴冷的角落（自动实现属性也属于这种情况）。幸运的是，尽管命名实参和可选参数不是专门为辅助不易变性而设计的，但使用它们可以编写类似对象初始化程序这样，可调用构造函数或工厂方法的代码。

例如，假设我们要创建一个Message类，那么来源（from）地址、目的（to）地址和消息体（body）是必备的，主题和附件是可选的（为了使示例尽可能简单，我们只设置一个接收者）。我们可以使用适当的可写属性创建一个易变的类型，并像下面这样构造实例：

```
Message message = new Message {
   From = "skeet@pobox.com",
   To = "csharp-in-depth-readers@everywhere.com",
   Body = "Hope you like the third edition",
   Subject = "A quick message"
};
```

这里有两个问题：首先，它并没有强制要求提供必备数据。我们可以强制构造函数提供这些数据，但这种情况下（C# 4之前）各个实参的含义就无法一目了然了：

```
Message message = new Message(
   "skeet@pobox.com",
   "csharp-in-depth-readers@everywhere.com",
   "Hope you like the third edition")
{
   Subject = "A quick message"
};
```

其次，这种初始化模式不能用于不易变类型。编译器在初始化对象之后才会调用setter属性。

但我们可以使用可选参数和命名实参，使其具备第一种形式（仅指定感兴趣的内容并提供名称）中的这些优秀特性，并且仍然对消息各部分是否必备进行验证，也不会丧失不易变性的好处。代码清单13-7展示了一种可能的构造函数签名以及构造步骤，这个消息与之前的消息相同。

代码清单13-7　使用C# 4构造不易变的消息

```
public Message(string from, string to,
               string body, string subject = null,
               byte[] attachment = null)
{
                                              ◁ 正常的初始化代码
}
...
Message message = new Message(
   from: "skeet@pobox.com",
   to: "csharp-in-depth-readers@everywhere.com",
   body: "I hope you like the third edition",
   subject: "A quick message"
);
```

我真的很喜欢这种代码的可读性和整洁性。你不需要设计大量用于选择的构造函数，只需要准备一个包含一些可选参数的构造函数即可。它与对象初始化程序也不同，相同的语法还可以用于静态创建方法。唯一的缺点是，你的代码必须由支持可选参数和命名实参的语言消费。否则调用者将不得不编写丑陋的代码来指定所有可选参数的值。显然，对于不易变性的支持，相比从初始化代码中取值来说还差得很远，但这仍然是向正确方向迈出的受欢迎的一步。

在开始COM之前，关于这些特性还有几点需要指出，即有关编译器如何处理代码，以及优良API设计的难点等问题。

2. 重载决策

显然，命名实参和可选参数都影响了编译器的重载决策——如果多个方法的签名相同，应该使用哪一个？可选参数会增加适用方法（applicable method）的数量（如果方法的参数数量多于指定的实参数量），而命名实参会减少适用方法的数量（通过排除那些没有适当参数名称的方法）。

在大多数情况下，这些改变都是很直观的：为了检查是否存在特定的适用方法，编译器会使用位置参数的顺序构建一个传入实参的列表，然后对命名实参和剩余的参数进行匹配。如果没有指定某个必备参数，或某个命名实参不能与剩余的参数相匹配，那么这个方法就不是适用的。7.5.3节[①]相关规范中详细介绍了这些内容，不过我认为有两种情况特别值得我们注意。

① 即《C#语言规范》的7.5.3节。——译者注

首先，如果两个方法均为适用的，其中一个方法的所有实参都显式指定，而另一个方法使用了某个可选参数的默认值，则未使用默认值的方法胜出。但这并不适用于仅比较所使用的默认值数量这种情况——它是严格按照“是否使用了默认值”来划分的。例如下面的代码：

```
static void Foo(int x = 10) {}
static void Foo(int x = 10, int y = 20) {}
...
Foo();                          ❶ 错误：会引起歧义
Foo(1);                         ❷ 调用第一个重载
Foo(y: 2);                      ❸ 调用第二个重载
Foo(1, 2);                      ❹ 调用第二个重载
```

在第一个调用中❶，由于两个方法的参数都是可选的，因此它们都是适用的。但编译器无法计算出你要调用的是哪个方法，因此将引发一个错误。在第二次调用中❷，两个方法仍然都是适用的，但使用的将是第一个重载，因为它没有使用任何默认值，而第二个重载使用了y的默认值。在第三次和第四次调用中，只有第二个重载才是适用的。第三次调用❸命名了y实参，第四次调用4包含了两个实参，这都意味着第一个重载是不适用的。

说明 **重载和继承并不总是能很好地混用** 所有这一切都是建立在编译器找到了多个可选重载的基础上的。如果某些方法声明在基类型中，而其派生类型中包含适用方法，那么后者将胜出。情况总是这样，并且会产生意想不到的结果（可登陆本书网站http://mng.bz/aEmE，了解更多相关内容）。而可选参数意味着适用方法比我们预计的要多。我建议如果没有特别的好处，应尽量避免在派生类中重载基类的方法。

其次，命名实参有时候可以代替强制转换，来辅助编译器进行重载决策。如果两个不同的方法都能将实参转换为参数类型，并且哪种方法都不比另一种方法更好，这时就会引起歧义。例如，考虑如下的方法签名和调用：

```
void Method(int x, object y) { ... }
void Method(object a, int b) { ... }
...
Method(10, 10);                 有歧义的调用
```

两个方法都是适用的，但哪一个都不比另一个更优。如果不想更改方法名称以消除歧义的话，那么有两种方式可以解决这个问题。（改名是我的首选方案。明确而详实的方法名称可以提升代码的可读性。）你可以显式强制转换某个实参，或使用命名实参：

```
void Method(int x, object y) { ... }
void Method(object a, int b) { ... }
...
Method(10, (object) 10);        通过强制转换消除歧义
Method(x: 10, y: 10);           通过命名实参消除歧义
```

当然，只有在方法不同且参数名称不同的情况下这种方法才有效，但这确实是一个方便的技巧。有时强制转换可以增加代码可读性，有时则是命名实参。这个方法只是我们为整洁代码而战的额外武器。

可惜，命名实参使用该方法时通常存在一个缺点。因此更改参数名称时，这也是一个需要注意的问题。

3. 恐怖默片：更改名称

在过去，如果你只使用C#的话，参数名称还没有显得那么重要。其他语言也许很在意名称，但在C#中，参数名称只有在使用智能感知或查看方法代码本身时才略显重要。而现在，即便只使用C#，方法的参数名称也已经成为高效API的一部分。如果日后修改了它们，可能会导致代码崩溃——如果修改某个参数的名称，那么任何指向该参数的命名实参都将编译失败。如果你的代码只有你自己使用，这可能也不是什么问题，但如果编写的是公共API，修改参数名称就是一件大事了。其实一直都是这样，只不过以前如果调用代码的也是C#，我们可以将其忽略，而现在则不同了。

修改参数名称已经很糟糕了，交换参数名称则更糟。那样的话调用代码也许仍然能够编译，但含义则完全不同了。覆盖某个方法并在覆盖版本中交换参数名称是其中尤其糟糕的一种情况。编译器总是根据作为方法调用目标的表达式的静态类型，查找它所知道的最深的覆盖版本。通过相同的实参列表，调用相同的方法实现，而根据变量的静态类型却得到了不同的行为，这绝对不是你希望看到的情况。

4. 小结

命名实参和可选参数大概是C# 4中听上去最简单的特性，但我们已经看到，它们仍然有很多复杂之处。其基本理念是易于表达和理解的，而且大多数时候这就是你需要关注的全部内容。我们可以利用可选参数来减少重载的数量，也可以使用命名实参为容易混淆的实参增加可读性。

最头疼的大概是决定使用哪个默认值，因为可能存在潜在的版本问题。同样，参数名称的问题也比之前更加明显，在覆盖已知方法时你需要特别注意，不要让你的行为成为调用者的噩梦。

说到噩梦，我们来看看与COM相关的新特性。好吧，这只是个玩笑。

13.2　改善 COM 互操作性

我得承认，自己与COM专家相去甚远。在.NET出现之前，我使用COM时常常会陷入困境，部分原因是我知识匮乏，而另一方面则是因为我所使用的组件设计和实现得都很差劲。这种总体感觉就像某种巫术一样挥之不去。尽管我确信COM有很多让人喜欢的地方，但遗憾的是我一直没有回过头去详细学习——而且真的是有很多细节需要学习。

说明　**本节只针对微软的编译器**　对COM互操作性的改善不会对所有C#编译器都有意义。即便其他编译器没有实现这些特性，也仍然是符合规范的。

总的来说，.NET使COM变得更加友好了。但在此之前，用Visual Basic操作COM要比C#有明显的优势。在本节你将看到，C# 4的出现使双方变得势均力敌。鉴于大家都对Word非常熟悉，本章我将使用它作为示例，下一章使用Excel。当然，这些特性不是只针对Office的。你会发现，无论你做什么，使用C# 4操作COM的体验都要比以前好得多。

13.2.1　在C# 4 之前操纵Word是十分恐怖的

我们的示例非常简单——启动Word，创建包含一小段文本的文档，保存然后退出。如果只有这些操作，听上去很容易，是吧？代码清单13-8展示了在C# 4之前所需的代码。

代码清单13-8　在C# 3中创建和保存文档

```
object missing = Type.Missing;

Application app = new Application { Visible = true };        <-❶ 启动Word
app.Documents.Add(ref missing, ref missing,         <-❷ 新建文档
                  ref missing, ref missing);
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add(ref missing);
para.Range.Text = "Thank goodness for C# 4";

object filename = "demo.doc";                    <-❸ 保存文档
object format = WdSaveFormat.wdFormatDocument97;
doc.SaveAs(ref filename, ref format,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing);

doc.Close(ref missing, ref missing, ref missing);      <-❹ 关闭Word
app.Application.Quit(ref missing, ref missing, ref missing);
```

该代码中的每个步骤都很简单：首先创建COM类型的实例❶，并使用对象初始化程序表达式将其设置为可见；然后创建并填充文档❷。向文档插入文本的机制可能不像我们预想的那样简单，要记住Word文档的结构相当复杂，但没有你看上去那样糟糕。几个方法调用都使用了可选的引用参数，我们对此不感兴趣，因此传递一个局部变量的引用，值为`Type.Missing`。如果你以前使用过COM，会对这些模式相当熟悉。

真正让人讨厌的是保存文档❸。是的，`SaveAs`方法包含16个参数，而我们只使用了两个。甚至连这两个也需要按引用传递，这意味着要为它们创建局部变量。从可读性角度来说，这简直是噩梦。不过别担心——稍后我们就来解决这个问题。

最后，关闭文档和应用程序❹。两个调用都包含3个我们不关心的可选参数，除此之外也就没什么了。

我们先来使用本章已学的特性——它们可以显著地减少示例的代码量。

13.2.2　可选参数和命名实参的复仇

首先，让我们去除那些毫不相关的可选参数所对应的实参。这也意味着我们不再需要`missing`变量。

不过`SaveAs`方法的16个参数仍然还剩下两个。根据本地变量的名称，很容易区分哪个参数是什么含义——但如果我们分配反了呢？所有的参数都是弱类型的，所以我们本质上还得进行猜

测。我们可以指定实参的名称来使调用更加清晰。如果我们还想使用后面的参数，也需要指定它们的名称，这样就可以跳过那些不感兴趣的参数。

代码清单13-9看上去已经整洁多了。

代码清单13-9　使用规范的C# 4特性操纵Word

```
Application app = new Application { Visible = true };
app.Documents.Add();
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add();
para.Range.Text = "Thank goodness for C# 4";

object filename = "demo.doc";
object format = WdSaveFormat.wdFormatDocument97;
doc.SaveAs(FileName: ref filename, FileFormat: ref format);

doc.Close();
app.Application.Quit();
```

这样就好多了，尽管为`SaveAs`指定实参时创建本地变量仍然是丑陋的。同样，如果你仔细阅读了前面的内容，可能会关心那些移除的可选参数。它们是`ref`且可选的参数，通常C#不支持这种组合。这是怎么回事呢？

13.2.3　按值传递`ref`参数

C#对`ref`参数的要求通常是很严格的。你需要将实参像参数那样标记为`ref`，来表示你明白这是怎么回事。所调用的方法可能会更改你声明的变量。在普通的代码中这都没有什么问题，但基于性能方面的考虑，COM API中几乎所有东西都使用了`ref`参数。而实际上它们通常不会修改传入的变量。在C#中将实参按引用传递是一件痛苦的事情。你不但要写上`ref`修饰符，还必须声明变量。你不能仅仅将值按引用传递。

C# 4编译器大大简化了这一过程，它允许你按值向COM方法传递实参，即使这是一个`ref`参数也是如此。考虑下面的调用，参数声明为`ref object`，而`argument`可能为`string`类型的变量：

```
comObject.SomeMethod(argument);
```

编译器将生成与下面等价的代码：

```
object tmp = argument;
comObject.SomeMethod(ref tmp);
```

注意，`SomeMethod`方法作出的任何修改都将被忽略，因此这个调用实际上是将`argument`按值传递。同样的过程也可用于可选的`ref`参数，即那些用`Type.Missing`初始化本地变量，然后按引用传递给COM方法的可选`ref`参数。反编译简化后的C#代码，会发现生成的IL是异常庞大的，包含了所有额外的变量。

现在我们可以对Word示例进行收尾工作了，如代码清单13-10所示。

13

代码清单13-10　将实参按值传递给COM方法

```
Application app = new Application { Visible = true };
app.Documents.Add();
Document doc = app.ActiveDocument;
Paragraph para = doc.Paragraphs.Add();
para.Range.Text = "Thank goodness for C# 4";
doc.SaveAs(FileName: "test.doc",                    <——按值传递的实参
           FileFormat: WdSaveFormat.wdFormatDocument97);
doc.Close();
app.Application.Quit();
```

如你所见，最终的代码要比之前的整洁多了。对于Word这样的API来说，你仍然需要使用Application和Document这些核心类型中略显复杂的方法、属性和事件，但至少代码要易读得多。

在介绍对C# 4部署的改善之前，我们还需要了解COM支持的最后一个方面，即有关源代码的改变。

13.2.4　调用命名索引器

C# 4支持的某些特性在很久之前Visual Basic就可以灵活自如地运用了，本节要介绍的就是其中之一。CLR、COM和Visual Basic都允许带参数的非默认属性——在C#中称为命名索引器（named indexer）。C# 4之前的版本不仅不允许直接声明自己的命名索引器①，而且不提供使用属性语法访问命名索引器的方法。C#中唯一可以使用的索引器被声明为类型的默认属性②。对于用Visual Basic编写的.NET组件来说，这不是什么大问题，因为通常不推荐命名索引器。但COM组件，如Office的COM组件，则广泛使用了命名索引器。C# 4允许我们调用命名索引器，但仍然不能在类型中声明它们。

说明　*术语再次冲突*　贯穿本节的术语“索引器”，在VB中称为含参属性（parameterized property）。CLI规范中称其为索引属性（indexed property）。无论叫什么名称，在IL中它都声明为一个属性，并且含有参数。普通的索引器（就C#而言）由类型的默认成员（或默认属性）定义——例如，StringBuilder的默认成员为Chars属性（包含一个Int32类型的参数）。而当我在这里谈论命名索引器时，所指的不是类型的默认成员，因此需要按名称引用它们。

再次以Word为例，我们这次来展示单词的不同含义。Word中的_Application类型定义了一个名为SynonymInfo的索引器，其声明如下：

```
SynonymInfo SynonymInfo[string Word,
                        ref object LanguageId = Type.Missing]
```

① 不管怎样，直接使用。你可以手动应用System.Runtime.CompilerServices.IndexerNameAttribute，但这是C#作为一门语言不了解的内容。

② 即使用关键字this声明的属性。——译者注

这不是有效的C#语法，因为它声明了命名索引器，不过好在它的意图很明确。索引器的名称为`SynonymInfo`，它返回一个`SynonymInfo`对象的引用，并且包含两个参数，其中一个是可选的。（本例中的索引器名称与返回类型的名称相同，这纯属巧合。）

`SynonymInfo`用来查找单词的含义，以及每个含义的同义词。代码清单13-11使用了该索引器，用三种不同的方法展示了三个不同单词的含义数量。

代码清单13-11　使用命名索引器展示同义词数量

```
static void ShowInfo(SynonymInfo info)
{
    Console.WriteLine("{0} has {1} meanings",
       info.Word, info.MeaningCount);
}
...
Application app = new Application { Visible = false };

object missing = Type.Missing;
ShowInfo(app.get_SynonymInfo("painful", ref missing));     ❶ 使用以前的C#语法

ShowInfo(app.SynonymInfo["nice", WdLanguageID.wdEnglishUS]);     ❷ 指定两个参数

ShowInfo(app.SynonymInfo[Word: "features"]);     ❸ 使用可选参数

app.Application.Quit();
```

我们看到即使没有命名索引器，仅使用之前的特性也是可以访问的❶。例如，我们本来可以调用`app.get_SynonymInfo("better")`并使用可选参数。但从❷和❸可以看出，索引器语法比`get_`调用要轻便得多。你可能会争辩说这应该是一个方法调用，或者`SynonymInfo`属性应该是无参的，并且返回包含适当默认索引器的集合。这也是C#设计者为什么没有完全支持命名索引器（包括在C#中声明命名索引器）的原因之一。但问题是，在Word中它已经是一个索引器了，所以以索引器的方式来使用它应该是合情合理的选择①。❷使用了13.2.3节中介绍的隐式`ref`参数特性，只是为了好玩儿，❸省略了可选参数并命名了另一个实参。

关于可选参数和索引器还有一个细小的变化：如果所有参数都是可选的，并且我们不想指定任何实参，则必须省略方括号。因此我们应该使用`foo.Indexer`，而不能写成`foo.Indexer[]`。所有这些都不仅适用于获取索引器，也适用于设置索引器。

截至目前，一切安好——但编写代码只是工作的一部分。我们通常还需要将其部署到其他机器上。同样，C# 4也简化了这一过程。

13

13.2.5　链接主互操作程序集

在操作COM类型时，我们使用的是为组件库生成的程序集。通常为主互操作程序集（PIA，Primary Interop Assembly），它是由发布者签名的规范COM库互操作程序集。你可以使用Type Library Importer工具（`tlbimp`）为自己的COM库生成这个程序集。PIA提供了一种正确的访问

① 展示其实际意义会显得更加有趣，但会导致互操作问题，不过这与本章无关。详细内容请浏览本书的网站。

COM类型的方式，这减轻了我们的工作，但从另一个角度来说，也是相当痛苦的。它们非常庞大，即使你只需要一小部分功能，也必须引用整个PIA。此外，部署计算机还必须和编译的计算机使用同样版本的PIA。由于正确的版本已经被部署，这就会导致尴尬的版本问题，从而无法再发布PIA。如果有许多版本可用，但是它们都公开了我们需要的功能，你可能不得不发布不同的代码版本来进行引用工作。

C# 4提供了一种完全不同的方法，不是像引用其他程序集那样对PIA进行引用，而是进行链接。在Visual Studio 2010及以上版本中，这是程序集引用属性中的一个选项，如图13-4所示。

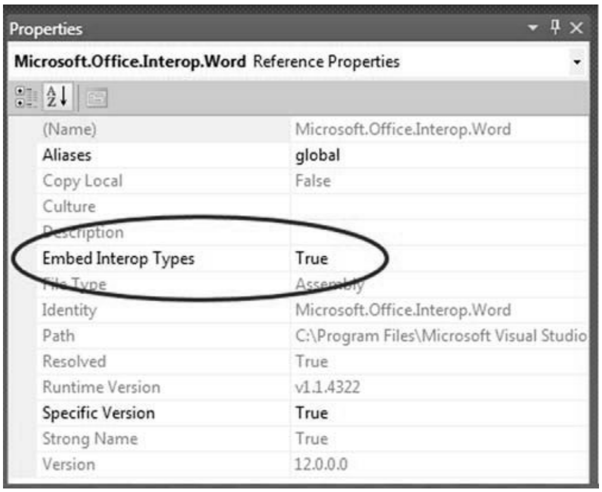


图13-4　在Visual Studio 2010中链接PIA

喜欢命令行的朋友可以使用`/l`（或`/link`）选项替换`/r`（或`/reference`），来实现对PIA的链接：

```
csc /l:Path\To\PIA.dll MyCode.cs
```

当链接一个PIA时，编译器只会将PIA中需要的那部分直接嵌入到我们自己的程序集中。它只获取我们需要的类型，以及这些类型中我们需要的成员。例如，编译器为本章编写的代码创建了以下类型：

```
namespace Microsoft.Office.Interop.Word
{
   [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
   public interface _Application

   [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
   public interface _Document

   [ComImport, CompilerGenerated, TypeIdentifier, Guid("...")]
   public interface Application : _Application

   [ComImport, Guid("..."), TypeIdentifier, CompilerGenerated]
   public interface Document : _Document
```

```
    [ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
    public interface Documents : IEnumerable

    [TypeIdentifier("...", "WdSaveFormat"), CompilerGenerated]
    public enum WdSaveFormat
}
```

而查看`_Application`接口，将看到代码如下所示：

```
[ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
public interface _Application
{
    void_VtblGap 1_4();
    Documents Documents { [...] get; }
    void_VtblGap2_1();
    Document ActiveDocument { [...] get; }
}
```

篇幅所限，我省略了GUID和属性特性，你可以使用Reflector查看这些内嵌的类型。它们都是接口和枚举——没有具体的实现。但一个普通的PIA会包含一个`CoClass`，来表示实际的实现（当然，这是真正COM类型的代理），当编译器需要通过链接的PIA创建一个COM类型的实例时，会使用与类型相关的GUID来进行创建。例如，Word示例中创建`Application`实例的那一行，当启用链接时将被翻译为以下代码[①]：

```
Application application = (Application) Activator.CreateInstance(
    Type.GetTypeFromCLSID (new Guid("...")));
```

图13-5展示了执行时的工作状况。

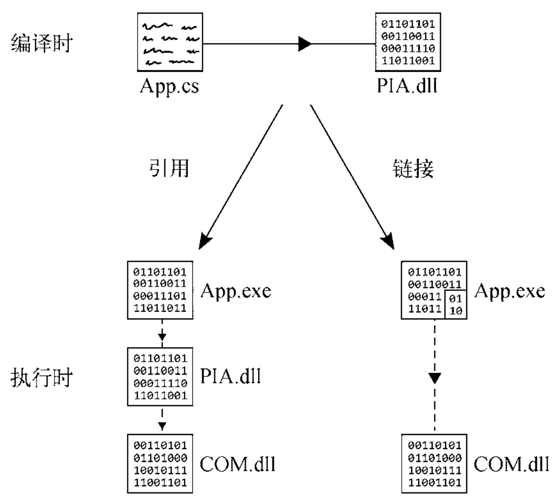


图13-5　引用和链接的比较

将类型库内嵌到程序集中有以下好处：

① 嗯，差不多吧。对象初始化器会使它稍微复杂一些，因为编译器使用了额外的临时变量。

- 易于部署：不需要原始的PIA，因此你不必依赖已经存在的正确版本，或自行发布PIA。
- 便于版本管理：只要实际安装的COM库的版本中包含我们使用的成员，就不必关心编译时使用的PIA版本。
- 变体（variant）被视为动态类型，以减少强制转换所需的开销。

对于最后一点现在不必担忧——解释了动态类型之后自然会拨云见日。所有悬念都将在下一章揭晓。

如你所见，微软在C# 4中认真地对待了COM互操作，使整个开发过程不再那么痛苦。当然痛苦的程度往往取决于所使用的COM库——有些库可以从新特性中受益，有些则不行。

下一个特性与COM、命名实参、可选参数完全无关，相同的一点在于它也简化了开发过程。

13.3　接口和委托的泛型可变性

你也许还记得，我在第3章中提到过CLR支持泛型类型的可变性（variance），但C#还没有公开支持。C# 4改变了这一点。C#获得了声明泛型变体所必需的语法，并且现在编译器也能知道接口和委托可能的转换。

这并不是什么带来翻天覆地的改变的特性——只是扫平了一些你偶尔会撞上的路障。它甚至并没有移除所有的障碍，并且还有各种各样的限制，主要是为了保持泛型的绝对类型安全。但这仍然是一个相当不错的特性。

你可能已经忘记了什么是可变性，我们首先简要介绍一下它的两种基本形式。

13.3.1　可变性的种类：协变性和逆变性

实质上，可变性是以一种类型安全的方式，将一个对象作为另一个对象来使用。我们已经习惯了普通继承中的可变性：例如，若某方法声明返回类型为`Stream`，在实现时可以返回一个`MemoryStream`。泛型可变性的概念与此相同，但要略微复杂一些。可变性应用于泛型接口和泛型委托的类型参数中，这一点必须引起注意。

最后，你能否记住本节术语并不是那么重要。阅读本章时它们可能很有用，但在平时谈话时你不大可能需要它们。概念才是最重要的。

可变性有两种类型：协变性和逆变性。二者概念基本相同，只是在上下文中转换的方向不同。我们先从协变性开始，它通常要好理解一些。

1. 协变性：从API返回的值

协变性用于向调用者返回某项操作的值。例如一个简单的表示工厂模式的泛型接口，它只包含一个方法`CreateInstanse`，返回适当类型的实例。代码如下：

```
interface IFactory<T>
{
   T CreateInstance();
}
```

现在，`T`在接口中只出现了一次（除了在签名中），它仅作为返回值使用，即方法的输出。这

意味着可以将特定类型的工厂视为更一般类型的工厂。如在现实世界里，你可以将比萨工厂视为食品工厂①。

2. 逆变性：传入API的值

逆变性则相反。它指的是调用者向API传入值，即API是在消费值，而不是产生值。我们来想象另一个简单的接口——它可以向控制台打印特定的文档类型。同样，它也只有一个方法Print：

```
interface IPrettyPrinter<T>
{
   void Print(T document);
}
```

这次T只作为参数出现在了接口的输入位置。具体而言，如果我们实现了IPrettyPrinter<SourceCode>，就可以将其当作IPrettyPrinter<CSharpCode>来使用②。

3. 不变性：双向传递的值

如果协变性适用于仅从API输出值的情况，而逆变性用于仅向API输入值的情况，那么如果值双向传递会如何呢？简而言之，什么也不会发生。这种类型是不变体（invariant）。

下面的接口表示可以对数据类型进行序列化和反序列化的类型：

```
interface IStorage<T>
{
   byte[] Serialize(T value);
   T Deserialize(byte[] data);
}
```

这时，如果存在一个具有特定类型T的IStorage<T>实例，我们不能将其视为该接口更具体或更一般类型的实现。如果以协变的方式使用（如将IStorage<Customer>视为IStorage-<Person>），则可能在调用Serialize时传入一个无法处理的对象③。类似地，如果以逆变的方式使用，则可能在反序列化数据时得到一个预料之外的类型④。

如果有助于理解的话，可以将不变性看成ref参数：按引用传递变量，其类型必须与参数本身的类型完全一致，因为值被传入了方法内部，并且同样被高效地传出。

13.3.2　在接口中使用可变性

在泛型接口或委托的声明中，C# 4能够使用out修饰符来指定类型参数的协变性，使用in修饰符来指定逆变性。声明完成之后，就可以对相关的类型进行隐式转换了。在接口和委托中，它们的工作方式是完全相同的，但为清晰起见，我们将分别介绍。我们先来介绍接口，它们似乎更

① 即存在从IFactory<Pizza>到IFactory<Food>的隐式转换。——译者注

② 即存在从IPrettyPrinter<SourceCode>到IPrettyPrinter<CSharpCode>的隐式转换。——译者注

③ 即如果存在从IStorage<Customer>到IStorage<Person>的隐式转换，那么当Serialize方法的参数为Student这样的非Customer类时，实际的IStorage<Customer>类将无法序列化这个Student。——译者注

④ 即如果存在从IStorage<Person>到IStorage<Customer>的隐式转换，那么当Deserialize方法返回Student类时，将无法得到期望的Customer。——译者注

常见，而且在前面描述可变性时我们已经使用了接口。

说明　变体的转换是引用转换　任何使用了协变和逆变的转换都是引用转换，这意味着转换之后将返回相同的引用。它不会创建新的对象，只是认为现有引用与目标类型匹配。这与在某个层次结构中，引用类型之间的强制转换是相同的：将一个Stream强制转换为MemoryStream（或反向的隐式转换），得到的仍然是同一个对象。稍后我们会看到，这样的转换会带来一些限制，但这其实意味着转换是有效的，并且有助于理解对象标识的行为

为了展示这些概念，我们将使用一些耳熟能详的接口，并用一些简单的自定义类型作为类型参数。

1. 用in和out表示可变性

我们使用的两个接口是IEnumerable<T>（对于T是协变的）和IComparer<T>（对于T是逆变的），它们可以很好地展示可变性。以下是它们在.NET 4中的声明：

```
public interface IEnumerable<out T>
public interface IComparer<in T>
```

这非常好记：如果类型参数只用于输出，就使用out；如果只用于输入，就用in。编译器可不知道你是否记住了哪种形式是协变，哪种形式是逆变。

可惜，框架并没有多少类（包含继承的层次结构）能够帮助我们非常清晰地展示可变性，因此我将退而求其次，使用几何形状这个标准的面向对象示例。可下载的源代码中包含IShape、Circle、Square的定义，它们的意思都很明确。接口的属性表示边框的形状和面积。在后面的示例中，我将频繁地使用两个列表，因此先将它们的构造代码展示如下，以供参考：

```
List<Circle> circles = new List<Circle>
{
   new Circle(new Point(0, 0), 15),
   new Circle(new Point(10, 5), 20),
};

List<Square> squares = new List<Square>
{
   new Square(new Point(5, 10), 5),
   new Square(new Point(-10, 0), 2)
};
```

唯一重要的地方在于变量的类型，我们声明的是List<Circle>和List<Square>，而不是List<IShape>。这通常是非常有用的，例如我们在其他地方访问圆形列表时，可以直接使用圆形特有的成员，而不必进行强制转换。构造函数中的数值是完全随意指定的。接下来我将直接使用circles和squares来引用这两个列表，而不必再复制代码[①]。

① 在整个源代码解决方案中，这两个属性是静态类Shapes的属性，但在代码段版本中，我在需要的地方引入了构造代码，这样就可以轻松地进行修改了。

2. 接口的协变性

为了演示协变性，我会基于圆形列表和方形列表来构建一个形状的列表。代码清单13-12展示了两种不同的方法，它们在C# 3下都无法运行。

代码清单13-12 根据圆形和方形列表构建一般形状的列表

```
List<IShape> shapesByAdding = new List<IShape>();       ❶ 直接在列表中添加
shapesByAdding.AddRange(circles);
shapesByAdding.AddRange(squares);
                                                        ❷ 使用LINQ进行连接
List<IShape> shapesByConcat = circles.Concat<IShape>(squares).ToList();
```

实际上，代码清单13-12在4个地方使用了协变性，对于类型系统而言，每次将圆形或方形序列转换为普通几何形状时，都用到了协变性。首先新建了一个`List<IShape>`，并调用`AddRange`向其添加圆形和方形列表❶。（我们也可以向构造函数传递一个列表，然后调用`AddRange`一次。）`List<T>.AddRange`的参数为`IEnumerable<T>`类型，因此在这种情况下我们将这两个列表都看成是`IEnumerable<IShape>`，而这在以前是不被允许的。`AddRange`可以设计为泛型方法，拥有自己的类型参数，但实际上没有这样做——这会导致某些优化难以甚至无法进行。

另一种根据已知序列的数据创建列表的方法是使用LINQ❷。我们不能直接调用`circles.Concat(squares)`，这会使类型推断机制变得混乱，而应显式地指定类型参数。`circles`和`squares`都将根据协变性而隐式转换为`IEnumerable<IShape>`。这种转换不会真正改变它们的值，所改变的只是编译器如何看待这些值。编译器不是在构建一个单独的副本，这一点相当重要。对于LINQ to Objects来说，协变性尤其重要，因为很多API都表示为`IEnumerable<T>`；而逆变性就没那么重要了，能涉及逆变性的类型十分有限。

在C# 3中，我们当然有其他途径可以解决这个问题。我们可以不通过原始形状列表`List<Circle>`和`List<Square>`来构建`List<IShape>`实例；我们可以使用LINQ的`Cast`操作符，将具体的列表转换为更一般的列表；我们也可以编写自己的列表类，包含泛型的`AddRange`方法。但没有一种方法能比之前的方法更加方便和高效。

3. 接口的逆变性

我们使用同样的几何形状类型来演示逆变性。这次我们只使用圆形列表，但却需要一个比较器来比较任意两个图形的面积。在C# 4之前，我们无法这样做，因为`IComparer<IShape>`不能用作`IComparer<Circle>`，但逆变性改变了这一切，如代码清单13-13所示。

13

代码清单13-13 使用通用的比较器和逆变性对圆形列表进行排序

```
class AreaComparer : IComparer<IShape>      ❶ 比较图形的面积
{
    public int Compare(IShape x, IShape y)
    {
        return x.Area.CompareTo(y.Area);
    }
}
...
IComparer<IShape> areaComparer = new AreaComparer();    ❷ 使用逆变性进行排序
circles.Sort(areaComparer);
```

这并不复杂。AreaComparer类❶差不多是最简单的IComparer<T>实现，例如，它不需要任何状态。在Compare方法中通常都会进行一些null处理[1]，但对于演示可变性来说，就没有必要了。

有了IComparer<IShape>，我们就可以用它对圆形列表进行排序❷。circles.Sort的参数应该为IComparer<Circle>类型，但逆变性会进行隐式转换。就是这么简单。

说明　意外，实在是令人感到意外　如果有人在C# 3下向你展示这段代码，你可能认为这可以运行。它看上去显然应该是能工作的，我们都会有同感。C# 2和C# 3中的不可变性往往是不受欢迎的"意外"。C# 4中这方面的新功能并没有引入新概念，它们只是增加了灵活性。

这两个示例都很简单，只使用了包含单一方法的接口，但同样的原则也可应用于更复杂的API。当然，接口越复杂，类型参数越有可能同时用于输入和输出，这使接口成为不变的。我们稍后将介绍一些复杂的示例，但首先先来看看委托。

13.3.3　在委托中使用可变性

我们已经学习了如何在接口中使用可变性，将同样的知识应用于委托就很容易了。我们将再次使用一些熟悉的类型：

```
delegate T Func<out T>()
delegate void Action<in T>(T obj)
```

它们实际上等同于我们一开始介绍的IFactory<T>和IPrettyPrinter<T>接口。使用lambda表达式，可以很轻松地进行演示，甚至可以将它们连接起来。代码清单13-14使用几何形状类型。

代码清单13-14　用Func<T>和Action<T>委托演示可变性

```
Func<Square> squareFactory = () => new Square(new Point(5, 5), 10);
Func<IShape> shapeFactory = squareFactory;        ❶ 使用协变性转换Func<T>

Action<IShape> shapePrinter = shape => Console.WriteLine(shape.Area);
Action<Square> squarePrinter = shapePrinter;      ❷ 使用逆变性转换Action<T>

squarePrinter(squareFactory());       <— 完整性检查
shapePrinter(shapeFactory());
```

现在这段代码不再需要任何解释了吧。我们的方形工厂总是生产位置相同且边长都为10的正方形。协变性允许我们将方形工厂视为更一般的形状工厂❶，这没有什么奇怪的。然后我们创建了一个通用的行为，打印任意形状的面积。这次我们使用逆变转换，让行为可用于任意方形❷。

---

[1] 即判断传入参数是否为null。——译者注

最后，我们将方形工厂的结果提供给方形行为（action），将形状工厂的结果提供给形状行为。结果都是预期的100。

当然，这里我们只使用了包含单一类型参数的委托。那么对于多个类型参数的委托和接口将会是什么情况呢？如果类型参数本身就是泛型委托，又会是什么情况呢？情况可能会变得很复杂。

13.3.4　复杂情况

在把你搞晕之前，我应该给你点安慰。尽管本节中我们要做的是非常奇妙的事情，但编译器会避免我们犯错。如果你以特别的方式使用了多种类型参数，可能仍然会被那些错误信息搞得不知所措，但只要编译通过，就是安全的了。委托和接口的可变性都可能会很复杂，尽管委托的版本常常会更简洁一些。让我们从一个简单的示例开始。

1. **Converter<TInput, TOutput>**：同时使用协变性和逆变性

.NET 2.0就已经包含了Converter<TInput, TOutput>委托类型。它与Func<T, TResult>是等效的，但意图更加明确。在.NET 4中，它变成了Converter<in TInput, out TOutput>，展示了哪个类型参数使用了哪种可变性。

代码清单13-15使用简单的转换器演示了可变性的一些组合。

代码清单13-15　用简单的类型演示协变性和逆变性

```
Converter<object, string> converter = x => x.ToString();
Converter<string, string> contravariance = converter;
Converter<object, object> covariance = converter;
Converter<string, object> both = converter;
```

❶ 将对象转换为字符串

❷ 将按钮转换为对象

代码清单13-15展示了委托类型Converter<object, string>（一个接收对象，生成字符串的委托）的可变性转换。我们首先使用简单的Lambda表达式（调用ToString❶）实现了委托。我们恰巧从未真正调用该委托，因此完全可以使用空引用。但我发现如果可以定义一个在调用时发生的具体行为，将有助于理解可变性。

接下来的两行代码相对简单，每次只需关注一种类型参数即可。TInput类型参数只用于输入，因此可以逆变地将Converter<object,string>当作Converter<Button,string>来使用。换句话说，既然可以将任意对象的引用传递给转换器，那么自然可以传递一个Button引用。同样，TOutput类型参数只用于输出（返回类型），因此可以协变地使用它：如果转换器总是返回一个字符串引用，那么在你能够保证返回一个对象引用的地方，可以放心地使用该转换器。

最后一行❷仅仅是对这种理念的一种合理延伸。它在同一个转换内同时使用了逆变性和协变性，得到的是一个只接受按钮并且返回对象引用的转换器。注意，如果不进行强制转换，则无法将其转换为原来的类型——我们已经基本上在每个点上都放松了要求，你不能再隐式地收紧了。

让我们再增加一点筹码，看看极端情况下能复杂成什么样。

2. 疯狂的高阶函数

当你将多个变体类型组合到一起时，真正怪异的事情就发生了。我不想在此深入过多的细节——只是想让你意识到这种潜在的复杂性。

让我们来看以下4个委托声明：

```
delegate Func<T> FuncFunc<out T>();
delegate void ActionAction<out T>(Action<T> action);
delegate void ActionFunc<in T>(Func<T> function);
delegate Action<T> FuncAction<in T>();
```

每一个声明都相当于将一个标准的委托嵌入到另一个之中。例如，`FuncAction<T>`等同于`Func<Action<T>>`，它们都表示一个函数，返回以`T`为参数的`Action`。但它应该是协变的还是逆变的呢？这个函数将返回与`T`相关的内容，似乎应该是协变的，但是它也传入了与`T`有关的内容，似乎又是逆变的。答案是该委托对`T`是逆变的，因此声明时使用了`in`修饰符。

作为一个便捷的规则，可以认为内嵌的逆变性反转了之前的可变性，而协变性不会如此。因此`Action<Action<T>>`对`T`来说是协变的，`Action<Action<Action<T>>>`是逆变的。相比之下，对于`Func<T>`的可变性来说，你可以编写`Func<Func<Func<... Func<T>...>>>`，嵌套任意多个级别，得到的仍然是协变性。

举一个使用接口的类似示例，假设可以使用比较器对序列进行比较。如果能够比较任意对象的两个序列，自然可以比较两个字符串序列，但反之则不然。将其转换为代码（不必实现接口）如下：

```
IComparer<IEnumerable<object>> objectsComparer = ...;
IComparer<IEnumerable<string>> stringsComparer = objectsComparer;
```

这种转换是合法的：由于`IEnumerable<T>`的协变性，`IEnumerable<string>`是比`IEnumerable<object>`更“小”的类型；而`IComparer<T>`的逆变性可以将“较大”类型的比较器转换为较小类型的比较器。

当然，我们在本节只使用了包含单个类型参数的委托和接口——它也完全可以应用于多个类型参数。尽管这种类型的可变性让你痛不欲生，不过不必担心，你并不会频繁地使用它，而且用的时候编译器也会帮你大忙。我只是想让你知道有这种可能。

另一方面，你可能认为某些功能可以实现，但实际上它们并没有得到支持。

13.3.5　限制和说明

C# 4对可变性的支持主要是受CLR的限制。对一门语言来说，支持底层平台禁止的转换是很困难的，这会导致一些奇怪的现象。

1. 不支持类的类型参数的可变性

只有接口和委托可以拥有可变的类型参数。即使类中包含只用于输入（或只用于输出）的类型参数，仍然不能为它们指定`in`或`out`修饰符。例如，`IComparer<T>`的公共实现`Comparer<T>`是不变的——不能将`Comparer<IShape>`转换为`Comparer<Circle>`。

除了实现方面的困难，从理论上看来也应该是这样的。接口是一种从特定视角观察对象的方式，而类则更多地植根于对象的实际类型。不可否认，继承可以将一个对象视为它继承层次结构中任何类的实例，由此在一定程度上削弱了这种理由的说服力。但不管怎样，CLR不允许这么做。

2. 可变性只支持引用转换

你不能对任意两个类型参数使用可变性，因为在它们之间会产生转换。这种转换必须为引用

转换。基本上，这使转换只能操作引用类型，并且不能影响引用的二进制表示。因此，编译器知道操作是类型安全的，并且不会在任何地方插入实际的转换代码。我们在13.3.2节提到过，可变转换本身是引用转换，所以不会有任何额外的代码。

特别地，这种限制禁止任何值类型转换和用户定义的转换。比如下面的转换是无效的。

- 将IEnumerable<int>转换为IEnumerable<object>——装箱转换；
- 将IEnumerable<short>转换为IEnumerable<int>——值类型转换；
- 将IEnumerable<string>转换为IEnumerable<XName>——用户定义的转换。

用户定义的转换比较少见，因此不成什么问题，但对值类型的限制可能会令你痛苦万分。

3. out参数不是输出参数

这曾让我大为诧异，尽管事后看来是有道理的。考虑使用以方法定义的委托：

```
delegate bool TryParser<T>(string input, out T value)
```

你可能会认为T可以是协变的——毕竟它只用在输出位置，是这样吗？

CLR并不真正了解out参数。在它看来，out参数只是应用了[Out]特性的ref参数。C#以明确赋值的方式为该特性附加了特殊的含义，但CLR没有。并且ref参数意味着数据是双向的，因此如果类型T为ref参数，也就意味着T是不变的。

事实上，即使CLR支持out参数，也仍然不安全，因为它可用于方法本身的输入位置；写入变量之后，同样也可以从中读取它。如果将out参数看成是"运行时复制值"似乎好一些，但它本质上是实参和参数的别名，如果不是完全相同的类型，将会产生问题。由于稍微有些繁琐，此处不再演示，但本书的网站上可以看到有关示例。

委托和接口使用out参数的情况很少，因此这可能不会对你产生影响，但为了以防万一，还是有必要了解的。

4. 可变性必须显式指定

在介绍表示可变性的语法时（即对类型参数使用in或out修饰符），你可能会问为什么要这么麻烦。编译器可以检查正在使用的可变性是否有效，因此为什么不能自动应用呢？

这样可以——至少在很多情况下是可以的——但我宁愿它不可以。通常我们向接口添加方法时，只会影响实现，而不会影响调用者。但如果声明了一个可变的类型参数，然后又添加了一个破坏这种可变性的方法，所有的调用者都会受影响。这会造成混乱不堪的局面。可变性要求你对未来发生的事情考虑周全，并且强迫开发者显式指定修饰符，鼓励他们在执行可变性之前做到心中有数。

对于委托来说，这种显式的特性就没有那么多争论了：任何对签名所做的影响可变性的修改，都会破坏已有的使用。但如果在接口的定义中指定了可变性的修饰符，而在委托声明中不指定，则会显得很奇怪，因此要保持它们的一致性。

5. 注意破坏性修改

每当新的转换可用时，当前代码都有被破坏的风险①。例如，如果你依赖于不允许可变性的

① 指新的API（可变性）可能会破坏已有代码。——译者注

13

is或as操作符的结果，运行在.NET 4时，代码的行为将有所不同。同样，在某些情况下，因为有了更多可用的选项，重载决策也会选择不同的方法。因此这也成了另一个显式指定可变性的理由：降低代码被破坏的风险。

这些情况应该是很少见的，而且可变性的优点也比潜在的缺点更加重要。你已经有了单元测试，可以捕获那些微小的变化，对不对？严肃地说，C#团队对于代码破损的态度非常认真，但有时引入新特性难免会破坏代码。

6. 多播委托与可变性不能混用

通常情况下，对于泛型来说，除非涉及强制转换，否则不用担心执行时遇到类型安全问题。不幸的是，当多个可变委托类型组合到一起时，情况就比较讨厌了。用代码可以更好地描述：

```
Func<string> stringFunc = () => "";
Func<object> objectFunc = () => new object();
Func<object> combined = objectFunc + stringFunc;
```

这段代码可以通过编译，因为将Func<string>类型的表达式转换为Func<object>是协变的引用转换。但对象本身仍然为Func<string>，并且实际进行处理的Delegate.Combine方法要求参数必须为相同的类型——否则它将无法确定要创建什么类型的委托。因此以上代码在执行时会抛出ArgumentException。

这个问题在.NET 4快发布的时候才被发现，但微软察觉到了，并且很可能会在未来的版本中予以解决（.NET 4.5中还未得到解决）。在此之前的应对之策是：基于可变委托新建一个类型正确的委托对象，然后再与同一类型的另一个委托进行组合。例如，略微修改之前的代码即可使其工作：

```
Func<string> stringFunc = () => "";
Func<object> defensiveCopy = new Func<object>(stringFunc);
Func<object> objectFunc = () => new object();
Func<object> combined = objectFunc + defensiveCopy;
```

庆幸的是，以我的经验来说，这种情况很少见。

7. 不存在调用者指定的可变性，也不存在部分可变性

与其他问题相比，这个问题的确更能引起你的兴趣，但值得注意的是，C#的可变性与Java系统相去甚远。Java的泛型可变性相当灵活，它从另一侧面来解决问题：不在类型本身声明可变性，而是在使用类型的代码处表示所需的可变性。

说明　**想了解更多内容吗？**　本书不是关于Java泛型的专著，但如果这个小插曲能引起你的兴趣，可以查看Angelika Langer的Java Generics FAQ（http://mng.bz/3qgO）。注意：这是一个异常庞大且错综复杂的话题！

例如，Java的List<T>接口大体上相当于C#的IList<T>。它包含添加和提取项的方法，这在C#中显然是不变的，而在Java中，你可以在调用代码时声明类型来说明所需的可变性。然后编译器会阻止你使用具有相反可变性的成员。例如，以下代码是完全合法的：

```
List<Shape> shapes1 = new ArrayList<Shape>();
List<? super Square> squares = shapes1;          // 声明为逆变的
squares.add(new Square(10, 10, 20, 20));

List<Circle> circles = new ArrayList<Circle>();
circles.add(new Circle(10, 10, 20));
List<? extends Shape> shapes2 = circles;         // 声明为协变的
Shape shape = shapes2.get(0);
```

我在很大程度上更倾向于C#泛型，而不是Java泛型。特别是类型擦除（type erasure）[①]在很多时候会让你痛苦万分。但我发现这种处理可变性的方式真的很有趣。我认为C#未来版本中不会出现类似的东西，所以你应该仔细考虑如何在不增加复杂性的前提下，将接口分割以增加灵活性。

在结束本章之前，还要介绍两处几乎是微不足道的改变——编译器如何处理`lock`语句[②]和字段风格的事件[③]。

13.4 对锁和字段风格的事件的微小改变

我不会对这两处改变投入太多篇幅，因为它们很可能永远不会影响到你。但如果你看过编译后的代码，并且想知道它们为什么会变成这样，了解一下这些内容还是很有帮助的。

13.4.1 健壮的锁

我们来考虑一小段使用了锁的C#代码。块中的细节不重要，但清晰起见我还是加了一条语句：

```
lock (listLock)
{
   list.Add("item");
}
```

在C# 4之前——包括使用C# 4处理.NET 4之前的东西时——以上语句将被有效地编译为下面的代码：

```
object tmp = listLock;        // ❶ 复制待锁定内容的引用
Monitor.Enter(tmp);           // 在try之前获取锁
try
{
   list.Add("item");
}
finally
{
   Monitor.Exit(tmp);         // 不管Add做了什么，都会释放锁
}
```

这没有问题，并且它还避免了一些问题。我们要确保释放的监视器与获取的是同一个，因此

① Java编译器使用类型擦除技术移除所有与类型参数有关的信息。你可以在Java的官方教程中（http://download.oracle.com/javase/tutorial/java/generics/erasure.html）了解更多关于类型擦除的内容。——译者注

② 关于lock语句的详细内容，可以参考C# 4.0规范的8.12节。——译者注

③ 关于字段风格的事件，可以参考C# 4.0规范的10.8.1节。——译者注

首先将被锁定内容的引用复制到一个临时局部变量内❶[①]。这同时意味着锁的表达式只会进行一次求值。然后我们在try语句块之前获取锁。因此如果获取锁的线程异常终止，则不会执行finally块中释放锁的语句。这还将导致另一个问题：如果线程在获取锁之后和进入try块之前异常终止，我们也无法释放锁。这可能会导致死锁——其他线程将一直等待该线程释放锁。尽管CLR一直以来都在努力阻止类似事情发生，但也不是完全没有可能发生。

我们所需要的，是一种原子地获取锁并知道它已经被获取的方式。幸运的是，.NET 4新增加了Monitor.Enter的重载，C# 4的编译器将使用这种方式：

```
bool acquired = false;
object tmp = listLock;
try
{
   Monitor.Enter(tmp, ref acquired);          <-- 在try块内部获取锁
   list.Add("item");
}
finally
{
   if (acquired)
   {                                          | 有条件地释放锁
      Monitor.Release(tmp);
   }
}
```

现在，当且仅当锁首先被成功获取时，才会被释放。

要注意在某些情况下，死锁并不是最糟糕的结果[②]；有时对于一个应用程序来说，让它继续运行比直接终止要更危险。但依赖于死锁条件是荒谬的；最好尽可能地避免线程异常终止。（终止当前执行线程是比较好的做法，这样可以得到更多的控制权。ASP.NET的Response.Redirect就是这么做的，但我仍然建议找到更好的控制流程的方式。）

在介绍C# 4中真正的重大特性之前，还有最后一个小的改变需要讨论。

13.4.2　字段风格的事件

值得简单一提的是，C# 4对字段风格事件的实现方式作了两处修改。尽管它们是潜在的破坏性更改，但似乎不会对你产生什么影响。

总之，字段风格的事件像字段一样进行声明，不再包含显式的add/remove语句块[③]，如下：

```
public event EventHandler Click;
```

首先，线程安全的实现方式发生了改变。在C# 4之前，字段风格的事件生成的代码锁定的是this（实例事件）或声明事件的类型（静态事件）。而C# 4中，编译器实现了线程安全，对原子的订阅和退订使用了Interlocked.CompareExchange<T>。与之前对lock语句的修改不同，

① 以避免其被更改。——译者注

② Eric Lippert有一篇介绍这一主题的优秀博文，标题是“Locks and exceptions do not mix”，网址是http://mng.bz/Qy7p。

③ 字段风格的事件编译之后，将包含一个后台字段（backing field），一个add方法和一个remove方法。——译者注

面对旧版本的.NET Framework时，这项更改同样适用。

其次，在声明事件的类中，事件名称的含义改变了。以前，在声明事件的类中订阅（或退订）事件——如`Click += DefaultClickHandler;`——将直接使用后台字段，完全跳过`add/remove`实现。现在情况变了，使用`+=`或`-=`时，事件的名称就指向事件本身，而不再是后台字段。当名称用于其他意图时（通常为分配或调用），则仍然指向后台字段。

尽管在平时使用时你可能不会注意这两处改变，不过它们是合理的，可以使一切变得整洁。Chris Burrows在他的博客中深入研究了这个话题，想了解更多内容可以参考http://mng.bz/Kyr4。

13.5 小结

本章在多个不同领域取精融汇（pick-and-mix）。话虽如此，但COM从命名实参和可选参数中受益良多，因此在介绍这两部分内容时，未免有内容上的重叠。

我认为C#开发者要想真正熟悉参数和实参的新特性，还需要一段时间。对于不支持可选参数的语言来说，重载仍然提供了额外的可移植性，而在你习惯之前，命名实参在某些情况下看上去仍然怪怪的。尽管如此，在演示构建不易变类型的示例中，好处还是显而易见的。在对可选参数分配默认值时需要引起一些注意，但我希望你使用`null`来作为"默认的默认值"，你会发现它是如此的有用和灵活，可以有效地避开一些可能遇到的限制和隐患。

对C# 4来说，处理COM已经走过了万水千山。我仍然倾向于使用纯粹的托管解决方案，只要能用，至少调用COM的代码现在已经易读多了，而且部署也得到了改善。我们没有完整介绍改进COM互操作的内容，因为下一章才会介绍对COM产生巨大影响的动态类型特性，但即便不考虑这一点，短小精悍的示例，寥寥几个步骤，也已经让我们兴奋不已了。

本章最后一个主要话题是可用于接口和委托的泛型可变性。有时你可能在对可变性毫无所知的情况下就使用了它，而我认为大多数开发者都更倾向于使用框架中声明的接口和委托的可变性，而不会自己创建它们。如果它偶尔看上去难以理解，我得说声抱歉，不过弄清它们到底是怎么一回事还是相当有好处的。可以聊以自慰的是，前C#团队成员Eric Lippert也在博客中公开承认高阶函数让他心烦意乱（参见http://mng.bz/79d8），我们都同命相连。关于可变性，Eric写了一个很长的系列（参见http://mng.bz/94H3），其中包括了很多有关设计决策的讨论。如果你现在对可变性的理解还有所欠缺，这将是一个非常不错的阅读资料。

为保持完整性，我们还快速浏览了C#编译器对如何处理锁和字段风格的事件所做的改变。

本章所介绍的是相对来说比较小的变化。第14章将介绍更加基础的内容：以动态方式使用C#的能力。

第 14 章

静态语言中的动态绑定

本章内容

- 何谓动态
- 如何使用C# 4中的动态类型
- COM、Python和反射的示例
- 如何实现动态类型
- 动态响应

一直以来，C#都是一门静态类型的语言。在某些情况下，C#编译器要寻找特定的名称而不是接口，如为集合初始化程序寻找适当的Add方法。然而在语言内部，却一直没有出现除了普通多态范畴之外的真正的动态性。这一点在C# 4中得到了改观——至少是部分改观。简单来说，我们有了一个新的静态类型dynamic，它可以在编译时做任何事，到执行时再由框架进行处理。当然，实际情况要复杂得多，这只是行动纲要。

由于在不使用dynamic的地方，C#仍然为静态类型语言，所以我并不期待那些动态编程的粉丝能一下子成为C#的拥趸。我们介绍C#并不是为了它的动态编程，而主要是为了它的互操作性。如果动态语言IronRuby和IronPython等都加入了.NET的生态系统中，而不能在IronPython中调用C#代码，那将是无法想象的，反之亦然。同样，在C#中调用COM API也常常很不方便，代码中充斥着大量的强制转换。动态类型解决了所有这些问题。另一方面，有大量项目使用C#内的动态类型，从而完成数据访问边界的简化。

本章中我会不断强调一点，即使用动态类型要十分小心。它研究起来很有趣，实现得也很好，但我还是建议你在大规模使用之前要深思熟虑。与其他特性一样，要权衡利弊，而不要仅仅因为它简洁（当然这一点是毫无疑问的）就盲目跟进。框架虽然出色地优化了动态代码，但它仍然在大多数情况下慢于静态代码。更重要的是，你失去了很多编译时的安全性。尽管单元测试可以在编译器无能为力的时候帮你找到很多突然出现的错误，但我还是更喜欢编译器的及时反馈，它可以告诉我正在使用的方法是否存在，或能不能用指定的参数进行调用。

另一方面，在某些情况下编译器所提供的安全级别也并不是很强。例如，使用反射时可能发生的错误要远远多于编译器能辨认的。比如，你通过方法名称调用某方法，但是该方法确实存在吗？我们的代码能访问它吗？提供的参数是否合适？编译器对于这些一点儿忙也帮不上。等价的

动态代码同样无法在编译时发现这些错误，但至少代码是相当易读且易懂的。我们要做的是用最合适的方法来解决所遇到的特殊问题。

动态行为在处理动态环境或数据时非常有用，但如果你真的希望编写大量动态代码，我还是建议你使用一门将动态性作为常规风格而不是例外情况的语言。C#仍是一门为静态类型而设计的语言，而那些一开始即设计为动态的语言常常含有大量特性，从而可使我们更有效地使用动态行为。由于现在可以简单地在C#中调用这些语言，我们就可以从那些静态类型运用起来如鱼得水的上下文中，将受益于动态风格的代码分离出来。

我并不想泼太多的冷水。在可以使用动态类型的地方，它还是要比其他选择更简单。在本章，我们会介绍C# 4中动态类型的基本规则，然后研究一些示例：动态地使用COM、调用IronPython代码以及简化反射操作。你可以在对这些细节一无所知的情况下执行这些操作，当我们尝到了动态类型的甜头之后，会继续研究后台到底发生了什么。我们还将特别讨论DLR和C#编译器面对动态代码时所做的工作。最后，我们还会学习如何使自定义的类型动态地响应方法调用、属性访问等。不过首先，我们要以退为进。

14.1 何谓、何时、为何、如何

在我们开始编写有关C# 4新特性的代码之前，先来看看为什么引入这个特性。我不知道还有什么其他的语言经历了从纯粹静态到部分动态的演变，不管你是频繁使用还是偶尔用之，它都是C#进化过程中十分重要的一步。

我们先来重新看看动态和静态的含义，然后考虑一些C#中动态类型的主要应用场景，最后研究其在C# 4中是如何实现的。

14.1.1 何谓动态类型

在第2章中，我介绍了类型系统的特征，并描述了C#是如何成为一门静态类型语言的。编译器知道代码中表达式的类型，知道任何类型中可用的成员。它应用了相当复杂的规则来决定哪个成员应该在何时使用。这包括了重载决策；在（动态类型出现）之前的唯一途径是根据对象在执行时的类型，来选择虚方法的实现。决定使用哪个成员的过程称为绑定（binding），对于静态类型的语言来说，绑定发生在编译时。

而在动态类型的语言中，所有的绑定都发生在执行时。编译器或解析器可以检查语法是否正确，但却无法检查所调用的方法或所访问的属性是否真的存在。这就好像一个没有字典的文字处理器：它能检查标点符号，却无法检查拼写是否正确。因此即使你对自己的代码特别自信，恐怕也需要一组良好的单元测试。一些动态语言通常都是解释型语言[①]，编译器不会参与其中。也有

14

① C#或Java这类语言通过编译器编译为机器码，然后直接在CPU中执行。而解释语言（interpreted language）则是通过解释器在执行时动态解释。详细内容可参考http://en.wikipedia.org/wiki/Interpreted_language。——译者注

一些语言同时提供了解释器和编译器，可以通过REPL（读取、求值、打印的循环）[①]来进行快速开发。

说明　**REPL和C#**　严格来说，REPL并不是动态语言所独有的。一些静态类型的语言也有在运行时进行编译的解释器。F#中的F# Interactive工具[②]正是如此。只不过对于动态语言来说，解释器更加常见。

C#也有类似的工具：Visual Studio的Watch and Immediate窗口中的表达式求值器（expression evaluator）可以看成是某种形式的REPL，而Mono中包含了一个C# Shell工具（参见http://mng.bz/nek9）。

值得一提的是，C# 4全新的动态特性不包含在执行时解释C#源代码的功能，例如不存在直接与JavaScript的`eval`等价的函数。要执行基于字符串数据的代码，需要使用CodeDOM API（特别是`CSharpCodeProvider`）或简单的反射来调用个别成员。此处也可选择Roslyn项目，不过截至本书撰写之时，该项目仍仅存在于社区技术预览版内。

当然，有时候，无论采取什么方式，都能完成同样的工作。由于让编译器在执行前进行了更多的准备工作，因此静态系统的性能往往比动态系统更优。鉴于以上这些缺点，你可能首先会质疑，为什么还会有人乐此不疲地使用动态类型呢？

14.1.2　动态类型什么时候有用，为什么

动态类型立足于两个有利要点。首先，如果你知道要调用的成员名称、要传入的参数以及要调用的对象，那么这就是我们所需要的全部了。这听上去像是你能拥有的全部信息，但C#编译器通常需要得更多。至关重要的是，为了准确地确定成员（模型重载），我们需要知道所调用的对象的类型和参数的类型。我们有时无法在编译时知道这些类型，即使你确实能保证代码运行时成员会存在并且正确。

例如，如果你知道正在使用的对象包含`Length`属性，那么至于它是`String`还是`StringBuilder`、`Array`、`Stream`还是其他包含该属性的类型，都是无关紧要的。你不需要将该属性定义在一些常用的基类或接口中，如果不存在这样的类型，它们可能会很有用。这种类型称为鸭子类型（duck typing），从概念上来说，“如果它走起来像鸭子、叫起来也像鸭子，那么就可以称之为鸭子。”[③]即使某个类型包含所需的全部内容，它也无法告诉编译器所谈论的确切类型。通过COM使用微软的Office API时就是如此。很多方法和属性的声明都返回`VARIANT`，这意味着调用

---

① REPL即read、evaluate、print loop，可理解为一种交互式提示符，是很多动态语言都具备的交互式编程环境，详见http://en.wikipedia.org/wiki/REPL。——译者注

② 详见http://msdn.microsoft.com/en-us/library/dd233175.aspx。——译者注

③ 维基百科上关于鸭子类型的词条包含更多关于该术语的历史信息：http://en.wikipedia.org/wiki/Duck_typing。

这些成员的C#代码常常夹杂着强制转换。鸭子类型允许你忽略所有这些转换，只要你对自己所做的事情有足够的信心。

动态类型的第二个重要的特性是，对象可以通过分析提供给它的名称和参数来响应某个调用。其行为就像是该类型正常地声明了成员一样，即使直到执行时我们才能知道成员的名称。例如，考虑如下的调用：

```
books.FindByAuthor("Joshua Bloch")
```

正常情况下，设计者应该在所涉及的类型中声明FindByAuthor成员。在动态数据层，会有一段智能代码来分析这种调用。它可以检测相关的数据（不管是来自数据库、XML文档、硬编码数据，还是其他什么地方）是否包含一个Author属性，并因此具备相应的行为。

在本例中，这意味着你要使用指定的参数作为作者来执行一个查询。从某种程度上来说，这只是下面代码的复杂形式：

```
books.Find("Author", "Joshua Bloch")
```

但第一段代码看上去更恰当，即使接收代码不知道Author部分，调用代码也会知道。这种方法可以在某些情况下用来模拟领域特定语言。也可以用来创建探索数据结构（如XML树）的原生API。

使用动态语言进行编程还有一个特性，正如我前面提到的，它往往是使用适当解释器进行编程的实验性风格。这一点并非与C# 4直接相关，但C# 4可以与运行在DLR（Dynamic Language Runtime，动态语言运行时）上的动态语言进行丰富的互操作，这意味着如果你要处理的问题可以从这种风格中受益，你就可以直接使用C#返回的结果，而不用后来再将其移植到C#中。

在学习完C# 4动态能力的基础之后，我们会深入研究以上情形，并将看到更多具体的示例。值得简要指出的是，如果这些优点对你并不适用，那么动态类型则很可能成为绊脚石，而不是助推器。很多开发者在日常编码时都不需要大量使用动态类型，即使确实需要，也可能只适用于一小部分代码。与其他特性一样，它可能被过度使用。是否具有其他设计方案，可以使用静态类型优雅地解决同样的问题？在我看来，仔细考虑一下这个问题总是有必要的。但由于我具备静态类型语言的背景，难免会有偏见——因此建议你阅读关于动态类型语言的书籍，如Python、Ruby等，来看看本章介绍之外它的诸多好处。

现在你应该会迫不及待地想看到一些真实的代码了吧，接下来我们对要发生的事情进行简要的介绍，然后深入研究一些示例。

14.1.3　C# 4 如何提供动态类型

C# 4引入了一个新的类型，称为dynamic。编译器对待该类型的方式与普通的CLR类型不同[①]。任何使用了动态值的表达式都会从根本上改变编译器的行为。编译器不会试图弄懂代码

① 事实上，dynamic并不代表一个特定的CLR类型。它实际上只是包含System.Dynamic.DynamicAttribute特性的System.Object。我们将在14.4节详细介绍，但现在你可以假装将其视为一个真正的类型。

的确切含义，不会恰当地绑定各个成员的访问，不会执行重载决策。它只是通过解析源代码，找出要执行的操作的种类、名称、所涉及的参数以及其他相关信息。编译器也不会发出（emit）IL来直接执行代码，而是使用所有必要的信息生成调用DLR的代码。剩下的工作将在执行时进行。

这在很多方面都与Lambda表达式转换成的不同种类的代码相类似。它们可以生成执行所需行为的代码（如转换为委托类型），也可以生成构建所需行为的描述的代码（如转换为表达式树）。稍后我们将看到，表达式树在DLR中是极其重要的，C#编译器常使用表达式树来描述代码。（最简单的情况，如果除了一次成员调用之外不再包含其他内容，就没有必要使用表达式树。）

当DLR在执行时绑定相关调用时，确定应该发生什么事情的过程非常复杂。在此期间，不仅仅要考虑方法重载等常规的C#规则，而且该对象本身也需要动态确定，如前面示例中看到的FindByAuthor。

这些大多发生在后台——你所编写的使用动态类型的源代码可以十分简洁。

14.2　关于动态的快速指南

还记得在学习LINQ时我们介绍了多少新的语法吗？动态类型则恰恰相反：只有一个上下文关键字dynamic，在可使用类型名称的地方，你差不多都可以使用该关键字。这就是新语法所要求的全部，关于dynamic的主要规则也很容易表述，希望你不会觉得以此作为开始会略显空洞。

- 几乎所有CLR类型都可以隐式转换为dynamic。
- 所有dynamic类型的表达式都可以隐式转换为CLR类型。
- 使用dynamic类型值的表达式通常会动态地求值。
- 动态求值表达式的静态类型通常被视为dynamic。

详细的规则会更加复杂，我们将在14.4节介绍，现在我们先使用简化的版本。

代码清单14-1演示了这些规则。

代码清单14-1　使用dynamic遍历列表，连接字符串

```
dynamic items = new List<string> { "First", "Second", "Third" };
dynamic valueToAdd = "!";
foreach (dynamic item in items)
{
   string result = item + valueToAdd;
   Console.WriteLine(result);
}
```

代码清单14-1的结果不会让你多么惊异：它将输出First!、Second!和Third!。在本例中，我们可以很容易地显式指定items和valueToAdd变量的类型，并且它们都将以正常的方式进行工作，但如果变量从其他数据源而不是硬编码中取值呢？如果向其中添加一个整数而不是字符串呢？

代码清单14-2略有变化，但valueToAdd的声明并未改变，改变的是它的赋值表达式。

代码清单14-2 动态地将整数与字符串相加

```
dynamic items = new List<string> { "First", "Second", "Third" };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
   string result = item + valueToAdd;          <-- String+int连接
   Console.WriteLine(result);
}
```

这时，第一个结果为First2，这正是你所期望的。使用静态类型，我们需要显式地将valueToAdd的声明由string改为int。尽管后面的加法运算符仍然会构建一个字符串。

如果我们将所有项都改为整数呢？让我们再做一处小的修改，如代码清单14-3所示。

代码清单14-3 整数与整数相加

```
dynamic items = new List<int> { 1, 2, 3 };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
   string result = item + valueToAdd;          <-- int+int相加
   Console.WriteLine(result);
}
```

悲剧！我们还在试图将相加的结果转换为字符串。而允许的唯一转换与C#中的常规语法并无二致[①]，因此不会将int转换为string。结果将产生异常（当然，是在执行时）：

```
Unhandled Exception:
   Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
Cannot implicitly convert type 'int' to 'string'
   at CallSite.Target(Closure , CallSite , Object )
   at System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet]
          (CallSite site, T0 arg0)
   ...
```

除非你已驾轻就熟，否则开始使用动态类型时会频繁遭遇RuntimeBinderException。从某种程度上说，它是一种新型的NullReferenceException。你肯定会不时地面对这个异常，但若幸运的话，它将出现在单元测试的上下文中，而不是用户的错误报告里。不管怎样，我们可以通过将result的类型修改为dynamic来解决这个问题，这样就不再需要转换了。

但细想一下，何苦一开始要使用结果变量呢？完全可以立即调用Console.WriteLine。代码清单14-4展示了这种修改。

代码清单14-4 整数与整数相加——没有异常

```
dynamic items = new List<int> { 1, 2, 3 };
dynamic valueToAdd = 2;
foreach (dynamic item in items)
{
   Console.WriteLine(item + valueToAdd);       <-- 调用以int为参数的重载
}
```

14

① 这里的意思是指字符串存在这种相"+"的语法操作。——译者注

不出所料可打印出3、4和5。改变输入数据，不仅会在执行时更改所选的操作符，而且还会改变所调用的Console.WriteLine重载。使用原始数据时，调用的是Console.WriteLine (string)；使用修改后的变量，调用的是Console.WriteLine (int)。数据甚至还可以包含混合的值，每次迭代将调用不同的重载。

dynamic还可用来声明类型的字段、参数和返回值。这与var形成了鲜明的对比，后者只能用于局部变量。

说明　**var和dynamic的区别**　在前面的很多示例中，如果我们在编译时就已经知道了确切的类型，就都可以使用var来声明变量。乍看上去，这两个特性十分相似。它们似乎都表示在声明变量时不指定类型，然而dynamic意味着我们显式地将类型设置为动态的。只有在编译器能够静态地推断出你声明的类型，并且整个类型系统仍然完全保持为静态时，才能使用var。当然，如果你用var所声明的变量是由dynamic类型的表达式初始化的，该变量的最终类型也将（静态地表示）为dynamic[①]。由于这会引起混淆，我强烈警告你不要这样做。

编译器对于它所记录的信息是非常智能的，在执行时使用这些信息的代码也十分智能：它可以称得上是一种迷你的C#编译器。它使用编译时得到的静态类型信息，使代码行为尽可能直观。

要在自己的代码中使用动态类型，除了无法使用动态类型实现的几个小细节之外，你真正需要了解的就是这些了。稍后我们将回过头来重新讨论这些约束，以及编译器真正处理的细节。首先让我们来看看动态类型都能做哪些真正有用的事情。

14.3　动态类型示例

动态类型有点类似不安全代码（unsafe code），或使用P/Invoke与本地代码交互。很多开发者可能用不到它，或很长时间才使用一次。对其他开发者来说——特别是那些处理微软Office的开发者——动态类型可以显著地提高生产力，无论是简化已有代码，还是以完全不同的方法来解决问题。

本节并不打算面面俱到。由于本书第二版已经出版，若干开源项目已使用动态类型并收到良好效果，如Massive（https://github.com/robconery/massive）、Dapper（http://code.google.com/p/dapper-dot-net/）和Json.NET（http://json.codeplex.com）。无论是与数据库对话，还是序列化与反序列化JSON，所有这些示例均位于数据边界。当然，但这并不意味着动态类型仅适用于数据边界，我也并不打算预测未来社区会出现什么新奇的用法。

这里我们将介绍3个示例：操作Excel、调用Python以及通过更灵活的方式使用普通托管.NET类型。

---

① 由编译器静态地推断为dynamic。——译者注

14.3.1 COM和Office

我们已经在第13章学习了大多数用于COM互操作的C# 4新特性，但有一个特性是无法在第13章介绍的，因为那时我们还没有学习动态类型。如果你将正在使用的互操作类型内嵌到程序集中（使用/l编译器开关，或将Embed Interop Types属性设置为true），那么该API中任何声明为object的东西[①]，都将变为dynamic。这大大简化了弱类型API（如Office所公开的API）的使用。（尽管Office中的对象模型在某种程度上是强类型的，但很多属性都公开为变体，可以处理数字、字符串、日期等。）

这里，我将再次展示一个简单的示例，它比第13章中的Word示例更加精短。在该示例中，动态部分很好理解。我们将一个新Excel工作表最顶行的前20个单元格的内容设置为数字1~20。代码清单14-5展示的是使用原始的静态类型实现的代码。

代码清单14-5 使用静态类型设置一个区域的值

```
var app = new Application { Visible = true };              ❶ 打开包含活动工作表的Excel
app.Workbooks.Add();
Worksheet worksheet = (Worksheet) app.ActiveSheet;
Range start = (Range) worksheet.Cells[1, 1];               ❷ 确定起始和结束单元格
Range end = (Range) worksheet.Cells[1, 20];
worksheet.Range[start, end].Value = Enumerable.Range(1, 20)
                                    .ToArray();             ❸ 用[1,20] 填充区域
```

本代码使用using指示符引入了Microsoft.Office.Interop.Excel命名空间（此处未显示），因此这里Application类型指向的是Excel，而不是Word。我们仍然使用C# 4的新特性，在调用Workbooks.Add()建立环境时，没有为可选参数指定实参❶，并且还使用了命名索引器❷。

当Excel启动并运行时，我们找出整个区域的起始和结束单元格。在本例中，它们虽然位于同一行，但我们可以通过选择两个对角创建一个矩形区域。你也可以只调用一次Range["A1:T1"]来创建区域，但我个人认为只使用数字会更加简单。像B3这样的单元格名称对人来说很好理解，但在程序中却难以使用。

有了区域之后，我们使用整型数组设置Value属性，以此来设置区域的值❸。由于我们只设置一行数据，因此可以使用一维数组；要设置跨多行的区域，需要使用矩形数组。

所有的代码都可以运行良好，但我们在区区6行代码中使用了3次强制转换。通过Cells和ActiveSheet属性调用的索引器都返回object。（许多参数也都声明为object类型，但这并没有太大影响，因为任何非指针类型都会隐式转换为object——只有进行相反的转换时，才需要强制转换。）代码清单末尾并没有关闭Excel，以便最后能够看到打开的工作表。

主互操作程序集会将所需的类型都嵌入到我们的二进制文件中，这样示例中所有的类型都将成为dynamic。如代码清单14-6所示，有了从dynamic到其他类型的隐式转换，我们可以移除所有的强制转换。

① 包括字段、参数、返回类型等。——译者注

代码清单14-6　在Excel中将dynamic隐式转换为其他类型

```
var app = new Application { Visible = true };
app.Workbooks.Add();
Worksheet worksheet = app.ActiveSheet;
Range start = worksheet.Cells[1, 1];
Range end = worksheet.Cells[1, 20];
worksheet.Range[start, end].Value = Enumerable.Range(1, 20)
                                              .ToArray();
```

除了强制转换，该示例与代码清单14-5完全一样。

需要注意的是，在执行时仍然会对转换进行检查。如果我们将start的声明类型改为Worksheet，转换将失败并抛出一个异常。当然，你不必非要执行这个转换。你可以用dynamic来表示所有的变量，如代码清单14-7所示。

代码清单14-7　全部使用dynamic

```
var app = new Application { Visible = true };
app.Workbooks.Add();
dynamic worksheet = app.ActiveSheet;
dynamic start = worksheet.Cells[1, 1];
dynamic end = worksheet.Cells[1, 20];
worksheet.Range[start, end].Value = Enumerable.Range(1, 20)
                                              .ToArray();
```

哪种方法更清晰呢？我是老式静态类型的粉丝，所以我选择代码清单14-6所示的方法。它在每一行都规定了我所期望的类型，这样如果有任何问题，就可以立即发现，而不必等到试图以某种可能得不到支持的方式使用某个值。

从最初开发时的生产力来说，这两种方法各有利弊。使用dynamic，我们不必得到期望的确切类型，而可以只使用它的值，并且只要它支持所有的操作，就不会有任何问题。另一方面，使用静态类型，还可以通过IntelliSense在任意阶段查看可使用的值。我们仍使用动态类型为Worksheet和Range提供隐式转换——我们只在一步中使用了一次，而没有大规模使用。从静态类型到动态类型的转换可能开始看上去并不太显眼，因为该示例比较简单，但随着代码复杂度的增加，消除了强制转换所带来的可读性也会随之增强。

从某种程度来说，COM是一项相对旧的技术，已是明日黄花。现在我们来与一些更新的技术进行交互，如IronPython。

14.3.2　动态语言

本节我只会使用IronPython作为示例[①]，当然它并不是唯一一门可用DLR进行交互的动态语言。可以说它是最成熟的，不过我们也有其他的选择，如IronRuby和IronScheme。DLR的一个既定目标是使新语言的设计者能够更加简单地创建出一门工作语言，与其他DLR语言、传统的.NET语言（如C#）都能很好地互操作，并且能访问庞大的.NET框架库。

① 要正确运行本节的示例，需要下载并安装IronPython，http://ironpython.codeplex.com/。——译者注

1. 为什么要在C#中使用IronPython

可能会有很多原因使你需要与动态语言进行互操作，就像.NET早期与其他托管语言互操作可以从中受益一样，VB开发者能够使用C#编写的类库，这显然很有用，反之亦然，那么动态语言为何不能如此呢？我询问过Iron Python in Action (Manning, 2009)的作者之一Michael Foord，让他列举了一些在C#应用程序中使用IronPython的场景，如下所示：

- 用户脚本；
- 在应用程序中用IronPython编写了一个层；
- 使用Python作为配置语言；
- 使用Python作为存储在文本（或数据库）中各种规则的规则引擎；
- 所使用的库在Python中可用，但.NET中没有此种库；
- 为调试而在应用程序中放入一个实时的解释器。

如果你仍持怀疑态度，可能会认为在主流应用程序中内嵌脚本语言的情况实属罕见——其实Sid Meier的电脑游戏[1]Civilization IV就可以使用Python来编写脚本。这不是后来才修改成这样的，电脑游戏核心设置的相当一部分都是用Python编写的。一旦构建了引擎，开发者会发现这个开发环境远比他们原来想象的要强大得多。

本章的示例将使用Python作为配置语言。与COM的示例类似，我会尽量保持简单，希望它能为你提供足够的初学体验。

2. 入门：内嵌Hello, World

如果想在C#应用程序中承载（host）或内嵌（embed）另一门语言，可以使用多种类型，这取决于你想得到的灵活性和控制度。此处我们的需求简单，只使用`ScriptEngine`和`ScriptScope`。本例中，我们会一直使用Python，所以可以让IronPython框架直接创建`ScriptEngine`。在更一般的情况下，可以使用`ScriptRuntime`通过名称来动态地选择语言实现。如果要求更高，你还可能需要使用`ScriptHost`和`ScriptSource`，以及其他类型的更多特性。

在最初的示例中，我们将打印`hello,world`两次，而不是一次。第一次将文本作为字符串直接传递给引擎，第二次从HelloWorld.py文件中加载。如代码清单14-8所示。

代码清单14-8　使用C#中内嵌的Python，打印`hello, world`两次

```
ScriptEngine engine = Python.CreateEngine();
engine.Execute("print 'hello, world'");
engine.ExecuteFile("HelloWorld.py");
```

基于同样的原因，你会认为这段代码既谈不上平淡无奇也说不上令人兴奋。它简单易懂，不需要什么解释。它也没做什么，只是简单地输出……然而让人振奋的是，在C#中内嵌Python代码是如此容易。当然，目前为止我们的互操作程度很低，但这确实已经简单到极限。

14

① 或生活方式，这取决于你如何看待现实世界和玩游戏的上瘾程度。

说明　Python中的多种字符串字面量形式　该Python文件中包含一行代码，即`print "hello, world"`。注意比较文件中的双引号与传入`engine.Execute()`方法字符串中的单引号。它们在各自的源中都是正确的。Python有多种字符串字面量表示法，如多行字面量可以使用三个单引号或三个双引号。之所以提及这些，是因为在将Python代码作为字符串字面量传入C#时，不需要再对双引号进行转义了。

下面要介绍的类型是`ScriptScope`，它对于配置脚本来说至关重要。

3. 用ScriptScope[①]存储和获取信息

以上使用的执行方法都包含一个重载，它们的第二个参数为作用域（scope）。简单来说，它可以看成是名称和值的字典。脚本语言在分配变量时常常不进行显式地声明，如果在程序最上层（而不是函数或类）这么做，通常会影响到全局作用域。

如果将`ScriptScope`实例传递给执行方法，那么引擎执行的脚本也将用于全局作用域。脚本可以在作用域中获取已经存在的值或创建新值，如代码清单14-9所示。

代码清单14-9　使用ScriptScope在宿主和脚本之间传递信息

```
string python = @"
text = 'hello'
output = input + 1
";
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
scope.SetVariable("input", 10);
engine.Execute(python, scope);
Console.WriteLine(scope.GetVariable("text"));
Console.WriteLine(scope.GetVariable("input"));
Console.WriteLine(scope.GetVariable("output"));
```

❶ Python代码作为C#字符串字面量嵌入到C#代码中
❷ 设置接下来使用的Python代码的变量
❸ 从作用域获取变量

本代码将Python源代码作为字符串逐字嵌入到C#代码中❶，而不再将其放入文件内，这样可以方便地在一个地方查看所有代码。我不建议你在产品代码中也这样做，部分原因是因为Python是对空格敏感的——一个看上去无害的格式修改，可能就会导致代码在执行时完全失败。

`SetVariable`和`GetVariable`方法显式地向作用域输入❷和获取值❸。正如你可能期望的那样，它们声明为`object`类型，而不是`dynamic`。但`GetVariable`允许你指定一个类型参数作为转换请求。

这与对非泛型方法的结果所做的强制转换并不完全相同，后者只是对值进行拆箱，这意味着必须将其转换为完全正确的类型。例如，我们可以将整数传递到作用域中，获取一个`double`类型：

```
scope.SetVariable("num", 20)
double x = scope.GetVariable<double>("num")
double y = (double) scope.GetVariable("num");
```

❶ 成功转换为double
❷ 拆箱操作将抛出异常

① `ScriptScope`的作用与命名空间类似。——译者注

第一个调用成功：我们显式地告诉GetVariable所需的类型❶，因此会强制返回适当的值。第二个调用❷会抛出InvalidCastException，与将值拆箱为错误类型时的情况一样。

作用域中也可以包含函数，我们可以动态地获取然后调用这些函数，传递参数并从中返回值。要做到这些，最简单的方式就是使用dynamic类型，如代码清单14-10所示。

代码清单14-10　调用声明在ScriptScope中的方法

```
string python = @"
def sayHello(user):
    print 'Hello %(name)s' % {'name' : user}
";
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
engine.Execute(python, scope);
dynamic function = scope.GetVariable("sayHello");
function("Jon");
```

配置文件通常并不需要这种功能，但在其他情况下却十分有用。例如，你可能会简单地使用Python为某绘图程序提供一个可在任何输入点调用的函数，以将其脚本化。在本书网站http://mng.bz/6yGi中可以找到这个简单的示例。

在很多情况下，能够在执行时运行用户输入代码的表达式求值程序是十分有用的，如计算折扣、运费的业务规则等。并且在以文本形式修改这些规则，而不用重新编译或重新部署二进制文件方面也非常有用。代码清单14-10很简单，可下载源代码中包含另一个示例，将两种语言复杂地交织在一起，展示了双向调用：我们已经看到的在C#中调用IronPython，以及在IronPython中调用C#[①]。

4. 综合应用

我们基本上已经介绍完了，因为已经可以从作用域中获取值。我们可以将作用域包装在另一个可通过索引器访问的对象里，甚至可以使用14.5节展示的技术来动态地访问值。程序代码可能会如下所示：

```
static Configuration LoadConfiguration()
{
   ScriptEngine engine = Python.CreateEngine();
   ScriptScope scope = engine.CreateScope();
   engine.ExecuteFile("configuration.py", scope);
   return Configuration.FromScriptScope(scope);
}
```

Configuration类型的具体形式取决于你的应用程序，不过这段代码好像不会让人特别兴奋。我已经在完整的源代码中提供了一个动态实现的示例，可以像属性一样获取值以及直接调用函数。当然，在配置文件中，我们不会局限于使用基元类型：Python代码可以十分复杂，可以构建集合、包装组件和服务等。它还能执行依赖注入或控制反转容器[②]的角色。

① 该示例位于OtherChapters/Chapter14目录下的PythonListComprehension.cs文件中。——译者注

② 关于依赖注入和控制反转，可以参考Martin Fowler著名的文章http://www.martinfowler.com/articles/injection.html。——译者注

重要的是，现在我们用一个积极的配置文件，替代了传统的消极的XML和.ini文件。当然，你可以在上面的配置文件中嵌入自己的编程语言，但那样效果可能不会太好，并且实现起来也会很费工夫。作为一个示例，比依赖注入更为简单的情形可能会更有用，你可能会为应用程序中的一些后台处理组件配置线程的数量。它们可能通常会与系统中处理器数目相同，但偶尔也会减少，以帮助另一个应用程序能够在同一系统中顺利运行。配置文件很可能从下面这种形式：

```
agentThreads = System.Environment.ProcessorCount
agentThreadName = 'Processing agent'
```

变为

```
agentThreads = 1
agentThreadName = 'Processing agent (single thread only)'
```

这种变化不需要重新生成或重新部署应用程序，而只需要编辑文件并重启应用程序。特别智能的应用程序甚至可以在运行时选择重新配置。（我发现这种功能实现起来的痛苦往往要大于它所带来的附加价值，但在某些地方也可以发挥巨大的作用。比如能够更改日志级别，无论对于特殊的代码段还是有困难的具体用户来说，这都可以使调试变得更加简单。）

除了执行函数，我们还没有真正看到如何以特定的动态方式使用Python。Python的全部功能都是可用的，并且在C#代码中使用dynamic类型可以充分利用元编程和所有其他动态特性的优势。C#编译器负责以适当的方式描述代码，脚本引擎负责接收代码并得出在Python中的结果。但不要因为在应用程序中嵌入脚本引擎很有价值，就以为自己做了一件多么聪明的事情。这只是迈向强大应用程序的一小步。

说明　**你想给脚本作者提供多大的权力？**　执行系统外部用户输入的恶意代码或特殊代码时，你需要认真地考虑安全问题，也许要在某种沙盒环境中执行脚本。该话题超出了本书的范围，但确实需要深思熟虑。

现在，我们的示例已经可以与其他系统进行互操作了。动态类型甚至可以在纯粹的托管系统中发挥作用。我们来看一些示例。

14.3.3　纯托管代码中的动态类型

几乎可以肯定的是，我们以前使用过一些类似动态类型的东西，即使执行代码并不是我们自己编写的。数据绑定就是其中最简单的示例——在为ListControl.DisplayMember这样的成员指定值时，我们实际上是要求框架在运行时根据其名称找到某个属性。如果在自己的代码中直接使用反射，你同样是在使用只有在执行时才可知的信息。

根据我的经验，反射很容易出错，即使可以运行，你也需要投入额外的精力对其进行优化。在某些情况下，动态类型可以完全取代反射，并且速度也可能更快（取决于实际所做的事情）。

在反射中使用泛型类型和泛型方法是十分复杂的。例如，某对象实现了基于某类型参数T的IList<T>，那么想得到T的具体类型是很困难的。如果得到T的目的是想调用另一个泛型方法，

那么你要求编译器调用的，就是在知道T的实际类型时将会调用的方法。当然，这就是动态类型所做的事情。我将使用该场景作为我们的第一个示例。

1. 执行时类型推断

如果你想要做的不仅仅是调用单个方法，那么最好将所有额外的工作包装在一个泛型方法内，然后动态地调用该泛型方法，而用静态类型来编写所有剩余的代码。代码清单14-11展示了这样一个简单的示例。

我们假设由系统的其他部分给定了某种类型的列表及一个新元素，它们是兼容的，但我们无法静态地知道它们的类型。这是完全有可能发生的——比如反序列化。无论如何，我们的代码是会在列表的元素个数少于10个时，将一个新的元素添加到列表的末尾。方法将返回元素是否添加成功。显然，现实生活中的业务逻辑会更加复杂，但重要的是我们希望在进行这种操作时使用强类型。代码清单14-11展示了静态类型的方法，以及对它的动态调用。

代码清单14-11　使用动态类型推断

```
private static bool AddConditionallyImpl<T>(IList<T> list, T item)
{
   if (list.Count < 10)                    ❶ 普通的静态类型代码
   {
      list.Add(item);
      return true;
   }
   return false;
}

public static bool AddConditionally(dynamic list, dynamic item)
{
   return AddConditionallyImpl(list, item);        ❷ 动态调用辅助方法
}
...
object list = new List<string> { "x", "y" };
object item = "z";
AddConditionally(list, item);          最终将调用AddConditionallyImpl<string>
```

公共的方法包含动态参数。在以前版本的C#中，参数可能为`IEnumerable`和`Object`，它们依赖复杂的反射检查来得出列表的类型，然后通过反射调用泛型方法。有了动态类型，我们就可以使用动态参数❷调用强类型的实现❶，用包装方法来分离对单个调用的动态访问。当然，该调用仍然可能失败，但我们已经不必费力地去确定适当的参数类型了。

我们也可以公开强类型的方法，以避免在知道列表为静态类型的情况下继续用动态类型。在这种情况下，应该让方法名称保持不同，以避免稍有不慎用参数的静态类型错误地调用方法的动态版本。（当名称不同时，对动态版本进行正确调用也会变得十分简单。）

在纯托管代码中使用动态类型的另一个示例，是我已经多次抱怨的在C#中缺乏支持的泛型操作符。你无法指定这样一个约束，即“T必须包含一个操作符，可以将两个T的值相加”。我们在最开始阐述动态类型时已经使用过类似的示例（详见代码清单14-4），因此这里再提及时你应该不会感到惊讶。我们使用LINQ中的`Sum`查询操作符，将其变为动态的版本。

2. 弥补泛型操作符的不足

你是否查看过Enumerable.Sum的重载方法列表呢？它相当长，并且不可否认的是，其中的一半用于投影，但即便这样重载也有10个之多，每个重载都只是获取一个元素序列，并将它们相加。而且这还不包括对无符号值、byte和short进行求和。为什么不用动态类型将它们放到一个方法里呢？

尽管我们将在内部使用动态类型，但代码清单14-12所示的方法仍然是静态类型的。我们可以将其声明为一个对IEnumerable<dynamic>求和的非泛型方法，但由于协变性的限制，这样并不能很好地工作[①]。我将方法名称由Sum改为DynamicSum，以避免与Enumerable中的方法冲突。因为如果两个方法的签名包含的参数类型相同，编译器将选择非泛型重载，而不是泛型重载[②]，而更改方法名称可以简单地在一开始就避免这种冲突。

代码清单14-12　动态地对任意元素的序列进行求和

```
public static T DynamicSum<T>(this IEnumerable<T> source)
{
   dynamic total = default(T);                          ❶ 即将使用的动态类型
   foreach (T element in source)
   {
      total = (T) (total + element);                    动态地选择其他操作符
   }
   return total;
}
...
byte[] bytes = new byte[] { 1, 2, 3 };
Console.WriteLine(bytes.DynamicSum());                  输出6
```

代码非常直接：几乎与其他普通的Sum重载的实现完全相同。简洁起见，此处忽略了检查source是否为空的语句，其余的大部分内容也十分简单，其中有几点非常有趣。

第一，我们使用default(T)来初始化total，它声明为dynamic，因此可以得到所需的动态行为❶。这样我们必须以某种方式使用一个初始值来开始操作；也可以使用序列中的第一个值，但如果序列没有元素就无法继续进行了。对于不可空的值类型，default(T)几乎总是一个适当的值：即自然数0。对于引用类型，我们会将序列中的第一个元素与null相加，这可能是恰当的，也可能不恰当。对于可空值类型，我们最终也会将第一个元素与该类型的空值相加，这肯定是不恰当的。

第二，我们将相加的结果强制转换回T，虽然接下来又将其赋值给了动态类型。这看上去可能很奇怪，但你想想两个字节求和的结果。C#编译器通常会在执行加法前将两个操作数提升为int。如果不强制转换，total变量将保存一个int值，当返回语句试图将其转换为byte时将抛出异常。

---

① 由于接口的协变性不能用于值类型，因此当使用接受参数为IEnumerable<dynamic>的方法对值类型序列（如IEnumerable<int>）求和时，会出现编译时错误。——译者注

② 比如，若将使用动态类型的方法声明为public static T Sum<T>(this IEnumerable<T> source)，那么当T为int时，将与Enumerable中已有的方法public static int Sum(this IEnumerable<int> source)冲突，编译器将选择后者，而不是前者。——译者注

这两点都将导致更深层次的问题，但这并不是本节的重点。我已经在本书网站上的一篇文章中对动态求和进行了详细的研究（参见http://mng.bz/0N37）。

此代码的适用范围不仅仅是对数字进行计算，代码清单14-13展示了如何对`TimeSpan`值求和。

代码清单14-13 对`TimeSpan`列表中的元素进行动态求和

```
var times = new List<TimeSpan>
{
    2.Hours(), 25.Minutes(), 30.Seconds(),
    45.Seconds(), 40.Minutes()
};
Console.WriteLine(times.DynamicSum());
```

方便起见，`TimeSpan`的值使用扩展方法创建，但求和过程是完全动态的，总跨度结果为3小时6分钟15秒。

3. 鸭子类型

有的时候，我们知道在执行时可以使用某个具有特殊名称的成员，但我们无法确切地告诉编译器这个成员，因为这将取决于具体的类型。从某种程度上，这是我们刚刚解决的那个问题的一个更普遍的示例，只是用普通的方法和属性替代了操作符。

一个区别之处是，通常我们可以捕获接口或抽象基类的共性。操作符无法实现这一点，但对于方法和属性来说却是十分常见的方法。而不幸的是，这并不总是行得通——特别是当涉及多个库的时候。.NET Framework基本上是一致的，但我们也看到过一个不一致的示例。在第12章，我们介绍了计算序列元素个数的优化算法，并且看到`ICollection`和`ICollection<T>`都包含`Count`属性——但它们却没有共同的包含该属性的父级接口，因此需要分别处理。

鸭子类型允许我们在访问`Count`时不必执行类型检查，如代码清单14-14所示。

代码清单14-14 使用鸭子类型访问`Count`属性

```
static void PrintCount(IEnumerable collection)
{
    dynamic d = collection;
    int count = d.Count;
    Console.WriteLine(count);
}
...
PrintCount(new BitArray(10));
PrintCount(new HashSet<int> { 3, 5 });
PrintCount(new List<int> { 1, 2, 3 });
```

`PrintCount`方法与集合初始化列表一样，限制参数必须实现`IEnumerable`，其原因也是相同的：它通常说明我们最终使用的`Count`属性是合适的。测试用的集合为`BitArray`（只实现了`ICollection`）、`HashSet<int>`（只实现了`ICollectin<int>`）和`List<int>`（两者都实现了），它们都可以在执行时找到正确的属性。

14

显式接口实现与动态类型不能混合使用

第一次测试这段代码的时候，我使用了int[]——它可以隐式转换为以上两种接口。但我惊讶地发现PrintCount方法在执行时失败了……后来我仔细考虑了这个问题。执行时绑定所使用的是对象的实际类型，在本例中为int[]。数组类型并没有公开Count属性——它们使用的是显式接口实现。你只能以一种特殊的方式使用数组的Count[①]。

动态类型的行为很可能看上去符合逻辑，但如果不仔细的话就会产生意想不到的结果，而以上只是其中一个示例。我在网站上将这些怪事收集成了一个持续更新的列表（参见http://mng.bz/5y7M），如果你发现了新的，可以告诉我。

我们后面还会继续这个获取子项数目的示例，但此时先来看看执行时的重载决策是如何为显式类型测试提供备选方案的。

4. 多重分发

对于静态类型，C#使用单一分发（single dispatch）：在执行时，所调用的确切方法只取决于覆盖（override）后的方法目标的实际类型。重载是在编译时确定的。多重分发（multiple dispatch）会根据执行时实参的类型，找出最适合的方法实现——同样，这也是动态类型所提供的。

代码清单14-15展示了使用动态分发如何为优化计数提供更加多样与健壮的实现。

代码清单14-15　使用动态分发有效地为不同的类型提供计数方法

```
private static int CountImpl<T>(ICollection<T> collection)
{
   return collection.Count;
}

private static int CountImpl(ICollection collection)
{
   return collection.Count;
}

private static int CountImpl(string text)
{
   return text.Length;
}

private static int CountImpl(IEnumerable collection)
{
   int count = 0;
   foreach (object item in collection)
   {
      count++;
   }
   return count;
}

public static void PrintCount(IEnumerable collection)
```

① 即将数组强制转换为ICollection。——译者注

```
{
   dynamic d = collection;
   int count = CountImpl(d);
   Console.WriteLine(count);
}
...
PrintCount(new BitArray(5));
PrintCount(new HashSet<int> { 1, 2 });
PrintCount("ABC");
PrintCount("ABCDEF".Where(c => c > 'B'));
```

我们知道，由于PrintCount的参数为IEnumerable类型，因此在执行时至少有一个CountImpl的重载是合适的。在代码清单12-17中，我们在选择随机元素时使用了这样的步骤：如果参数为ICollection<T>，就使用这个实现，如果为ICollection，就使用那个实现。在这里，我们依靠动态类型来执行相同的工作。代码清单14-15所示示例不仅使用了可用的Count属性，还为字符串做了优化，可以使用Length属性快速获得正确的结果。

即使在这里使用了多重分发，执行时仍然会遇到问题：如果实际类型通过显式接口实现，既实现了ICollection<string>又实现了ICollection<int>，该怎么办呢？选择不同的Count实现，将会产生两种可能的结果。在这种情况下，绑定会产生歧义，导致异常。幸亏这种病态情况非常罕见。

即使不与其他特性进行互操作，这些也是使用动态类型需要注意的方面，以上只是几个示例。接下来，在实现我们自己的动态行为并结束本章之前，我们来深入看看这些结果是如何产生的。

我要提醒你的是，事情将会变得很复杂。实际上，一切都很优雅，但也很复杂，因为编程语言提供了一组丰富的操作，而将这些操作的所有必要的信息表示为数据，然后恰当地执行，是非常复杂的工作。好消息是，你不必对所有东西都十分熟悉。与以往一样，对动态类型了解得越多，你就会越熟悉它的背后机理，但即使你只将其使用到目前这种程度，它也能大幅提高生产力。

14.4　幕后原理

鉴于前面我们提出的警告，我不会深入动态类型的大量内部细节。无论是框架还是语言方面的改变，都涉及很多内容。我并不会经常对规范的细节望而却步，但这次我确信即使全部学习也不会有很多收获。我会涵盖最重要（也是最有趣）的方面，如果你要深入研究某个特定场景，我强烈推荐你参考Sam Ng的博客（http://blogs.msdn.com/b/samng/）、C#语言规范和DLR项目（http://mng.bz/0M6A）以获取更多的信息。

我的最终目标是帮助大家弄清楚C#编译器做了什么，以及为了在执行时实现动态绑定所生成的代码。不幸的是，在搞明白支持这一切的机制（DLR）之前，查看生成的代码没有任何意义。静态类型的程序就像根据固定剧本上演的传统舞台剧，而动态类型的程序更像即兴表演。DLR取代了演员的大脑，可以根据观众的意见迅速作出响应。让我们来见识一下这位思维异常敏捷的明星。

14.4.1　DLR简介

我这段时间反复提到DLR这个术语，偶尔将其展开为动态语言运行时（Dynamic Language Runtime），但却从未进行过解释。我这是刻意为之：我一直试图讲清楚动态类型的性质，以及它如何影响开发者，而没有涉及任何实现的细节。但这个借口不可能持续到本章结束，因此此处要进行一下解释。说白了，动态语言运行时是所有动态语言和C#编译器用来动态执行代码的库。

令人震惊的是，它真的仅仅只是一个库。尽管它同样以运行时为名，却与CLR（Common Language Runtime，公共语言运行时）不在同一个级别——它不涉及JIT编译、本地API封送（marshal）、垃圾回收等内容。而是建立在大量.NET 2.0和.NET 3.5的功能之上，特别是`DynamicMethod`和`Expression`类型。.NET 4还扩展了表达式树的API，DLR可以使用它来表示更多的概念。图14-1将这些组织在了一起。

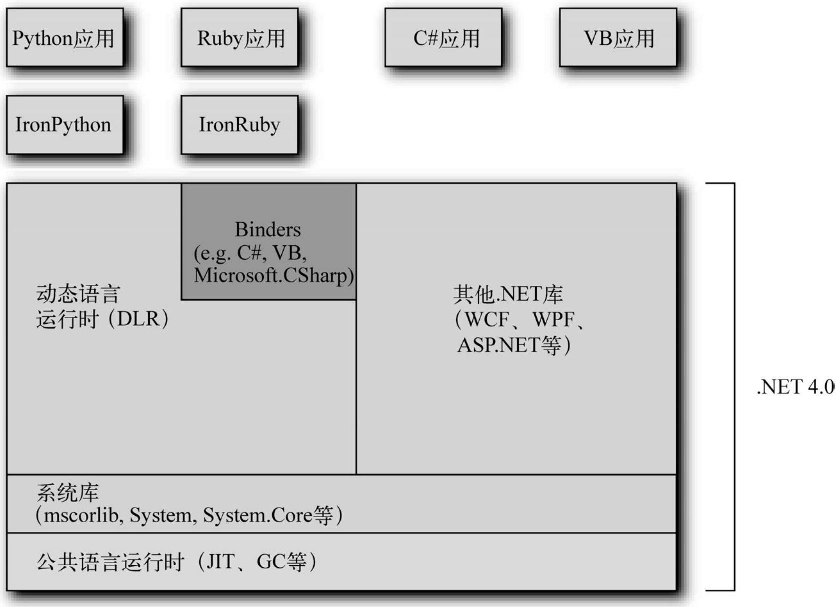


图14-1　.NET 4中的组件结构，允许静态和动态语言在相同的基础平台执行

除了DLR，图14-1还展示了另一个对你来说可能是全新的库。图中的binders部分有个程序集为`Microsoft.CSharp`。我们在代码中使用`dynamic`时，C#编译器所引用的很多类型都来自该程序集。令人不解的是，它并不包含已知的`Microsoft.CSharp. Compiler`和`Microsoft.CSharp. CodeDomProvider`。（这两个类型甚至彼此不在同一个程序集内！）我们将在14.4.2节学习这些新类型的用法，届时将反编译一些用`dynamic`编写的代码。

DLR与.NET Framework的其他部分的另一个重要区别是：它是开源的。完整的代码位于CodePlex项目中（http://dlr.codeplex.com），可以下载并研究其内部机制。这么做的好处之一是，Mono（http://mono-project.com）不用重新实现DLR了：它与.NET运行的代码相同。

尽管DLR不直接操作本地代码（native code），但在某种程度上我们可以认为它做着与CLR类似的工作：正如CLR将IL（中间语言）转换为本地代码一样，DLR将用绑定器、调用点（call site）、元对象（metaobject），以及其他各种概念表示的代码转换为表达式树，后者可以被编译为IL，并最终由CLR编译为本地代码。图14-2展示了一个经简化的动态表达式单次求值的生命周期。

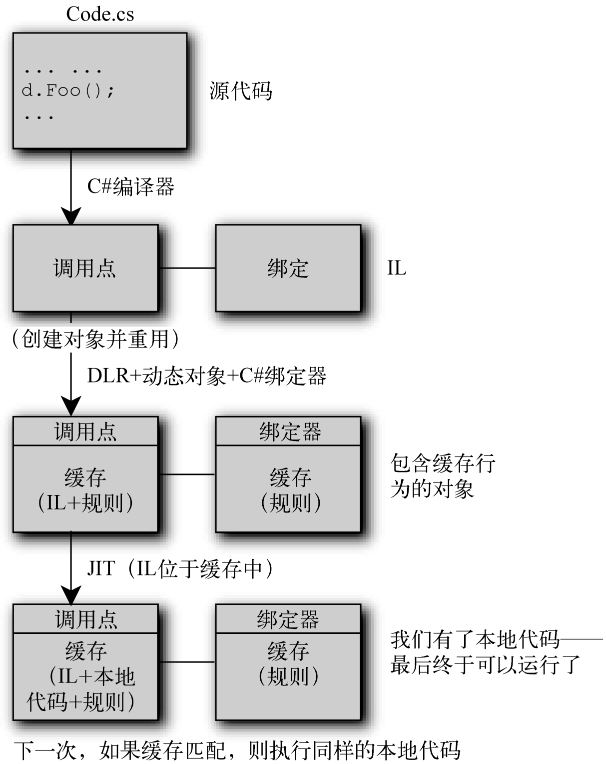


图14-2 动态表达式的生命周期

如你所见，DLR一个很重要的部分是多级缓存（multilevel cache）。这对性能而言十分重要，但为了理解这一点以及其他已经提到过的概念，我们需要从更深一级着手。

14.4.2 DLR核心概念

DLR的目的可以非常笼统地概括为，基于执行时才能知道的信息以高级形式表示并执行代码。本节我将介绍大量术语，来描述DLR是如何工作的，不过所有内容都将服务于这个目标。

1. 调用点

我们要介绍的第一个概念是调用点[1]。它有点像是DLR的原子——可以被视为单个执行单元

① 即调用方法的地方，我们称为“调用点”。参见http://en.wikipedia.org/wiki/Call_site。——译者注

的最小代码块。一个表达式可能包含多个调用点，但其行为是建立在固有方式之上的，即一次对一个调用点求值。

对于以后的讨论，我们只考虑单个调用点。对于调用点来说，例子越小巧越好，下面就是一个简单的示例，此处的d是一个dynamic类型的变量：

```
d.Foo(10);
```

调用点在代码中表示为一个System.Runtime.CompilerServices.CallSite<T>。我们将在下一节介绍C#编译器的功能时，看到如何创建并使用调用点的完整示例。下面的示例是在创建上面的站点时可能会调用的代码：

```
CallSite<Action<CallSite, object, int>>.Create(Binder.InvokeMember(
  CSharpBinderFlags.ResultDiscarded, "Foo", null, typeof(Test),
  new CSharpArgumentInfo[] {
   CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null),
   CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.Constant |
                             CSharpArgumentInfoFlags.UseCompileTimeType,
                             null) }));
```

现在我们有了一个调用点，可以执行代码了吗？貌似还不行。

2. 接收器和绑定器

除了调用点之外，我们还需要其他信息来判断代码的含义以及如何执行。在DLR中，有两个实体可以用来进行判断：接收器（receiver）和绑定器（binder）。调用的接收器是被调用的成员所在的对象。在我们的调用点示例中，接收器是执行时d所引用的对象。绑定器取决于调用语言，并且是调用点的一部分——在本例中，我们可以看到C#编译器生成的代码使用Binder.InvokeMember来创建绑定器。这里的Binder类为Microsoft.CSharp.RuntimeBinder.Binder，所以为C#特定的绑定器。C#绑定器也可用于COM，当接收器为IDispatch对象时将执行适当的COM绑定。

DLR总是将更高的优先级赋予接收器：如果动态类型知道如何处理调用，则将会使用该对象提供的执行路径。一个对象如果实现了新的IDynamicMetaObjectProvider接口，就具备了动态特性。名字虽然有点拗口，但他只包含一个成员：GetMetaObject。要正确实现GetMetaObject，需要成为一个表达式树高手并熟练掌握DLR。它同时也是一个强大的工具，可以与DLR及其执行缓存进行低级别的交互。如果你需要实现高性能的动态行为，花时间学习其细节是相当有必要的。

框架中有两个IDynamicMetaObjectProvider的公共实现，在性能不是很重要的情况下，它们可以方便地实现动态行为。14.5节将详细介绍所有这些内容，现在只需要认识到该接口本身以及它代表了对象动态响应的能力。

如果接收器不是动态的，绑定器将判断代码应该如何执行。在C#的代码中，它将对代码应用C#特定的规则并决定下一步做什么。如果你是在创建自己的编程语言，可以实现自己的绑定器，来决定其一般情况下的行为（如果该对象没有重载这个行为的话）。这完全超出了本书的范围，但其本身和相关内容却是非常有趣的话题：DLR的目标之一就是让你能够轻松地实现自己的语言。

3. 规则和缓存

如何执行一个调用所作出的决策，称为规则（rule）。从根本上来说，它包含两个逻辑元素：调用点表现为这种行为时所处的环境以及行为本身。

前者用于优化，比如某调用点将两个动态的值相加，并且在第一次求值时，两个值均为`byte`类型。绑定器进行了大量的工作，计算得出两个加数都应该提升为`int`，并且其结果应为这两个整数之和。每当两个加数均为`byte`时都会进行重用。对之前结果的有效性进行判断，可以节省很多时间。我用来作为示例的规则（两个加数的类型必须与上面完全一样）很常见，DLR还支持其他一些规则。

规则的第二部分是当规则匹配时所使用的代码，它表示为一个表达式树。它也可以存储为一个编译好的供调用的委托，但使用表达式树意味着可以对缓存进行深度优化。DLR中的缓存包含3个级别：L0、L1、L2。缓存以不同方式将信息存储于不同的作用域中。每个调用点包含自己的L0和L1缓存，而L2缓存可能被多个类似的调用点共享，如图14-3所示。

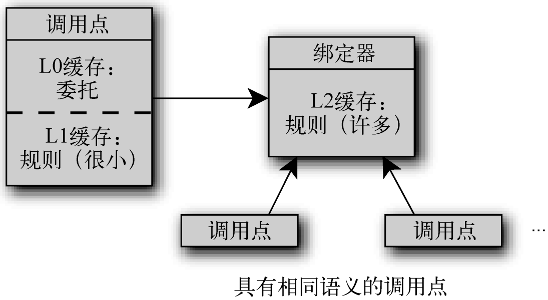


图14-3 动态缓存与调用点的关系

共享L2缓存的调用点是由它们的绑定器决定的——每个绑定器都包含一个与之相关的L2缓存。编译器（或其他创建了调用点的东西）决定要使用多少个绑定器。它可以只对多个表示类似代码的调用点使用一个绑定器，如果执行时的上下文相同，这些调用点应该以相同的方式执行。事实上，C#编译器没有使用这个功能——它会为每个调用点都创建一个新的绑定器①，因此对于C#开发者来说，L1和L2没有太大区别。但真正的动态语言，如IronRuby和IronPython都进一步使用了该功能。

缓存本身是可执行的，这可能不太好理解。C#编译器生成代码来简单地执行调用点的L0缓存（通过`Target`属性访问的委托）。就是这样！L0缓存包含单一的规则，将在调用时进行检查。如果规则匹配，将执行相关的行为。如果不匹配（或如果为第一次调用，还没有任何规则），将调用L1缓存，继而调用L2缓存。如果L2缓存找不到任何匹配的规则，将要求接收器或绑定器来解决这个调用。其结果将被放入缓存供以后使用。

对于前面的代码段，它的执行部分如下所示：

```
callSite.Target(callSite, d, 10);
```

① 很多信息都是特定于某个特殊的调用点的，因为绑定规则取决于诸如调用所在的类等事情。

L1和L2缓存以相当标准的方式审核它们的规则——每级缓存都包含一组规则，每条规则都会检查是否匹配。L0缓存则略有不同。行为的两个部分（检查规则和委派给L1缓存）将被合并为单独的方法，然后进行JIT编译。对L0缓存进行更新时将根据新的规则重新构建方法。

所有这些的结果是，处于相同上下文的调用点重复执行时的速度非常之快。如果手动编码进行测试，分发机制可变得十分简单。当然，还要权衡所有动态代码生成时所造成的损耗，但多级缓存恰恰是复杂的，因为它试图在多种场景中达到平衡。

我们已经对DLR的机制多少有了一点了解，现在就可以理解C#编译器到底做了什么，才能在运行时建立这一切。

14.4.3　C#编译器如何处理动态

在面对动态代码时，C#编译器的主要工作是解决什么时候需要动态行为，以及获取所有必需的上下文，这样绑定器和接收器在执行时就有足够的信息来处理调用。

1. 如果使用了动态，那么它就是动态的

如果所调用的成员的目标是动态的，那么这种情形显然就是动态的。编译器无从知道应该如何处理这种情况。它可能是真正的动态对象，将自行执行解决方案，也可能最终由C#绑定器通过反射来解决。不论是哪种情况，都已经根本没有机会来静态地解决这个调用了。

但如果调用的参数为动态的值，还是有可能进行静态处理的——特别是当某个重载包含适当的dynamic类型的参数时。其规则为，如果调用的任何一部分为动态的，该调用即为动态的，并将以动态值的执行时类型来匹配重载。代码清单14-16演示了这一点，该方法包含两个重载，并将以不同的方式进行调用。

代码清单14-16　方法重载和动态值

```
static void Execute(string x)
{
   Console.WriteLine("String overload");
}

static void Execute(dynamic x)
{
   Console.WriteLine("Dynamic overload");
}
...
dynamic text = "text";
Execute(text);                 <—— 打印"String Overload"
dynamic number = 10;
Execute(number);                <—— 打印"Dynamic Overload"
```

这两处对于Execute的调用都将进行动态绑定。在执行时，将使用实际值的类型进行处理，即string和int。除了在方法内部以外，dynamic类型的参数将被视为object类型——如果查看编译后的代码，将发现它确实为object类型的参数，只不过应用了额外的特性[①]。这还意味着

① 即DynamicAttribute特性。——译者注

在你声明的方法中，它们的签名不能只以dynamic/object参数类型来进行区分。

该示例处理方法调用，还应考虑众多其他的表达式。有时情况并不像你想象的那么简单。

2. 它是动态的……除非例外情况

在14.2节介绍dynamic时，我需要格外注意不能概括得太多，因为几乎所有规则都有例外情况。尽管你应该知道它们，但却没必要担心它们——它们不会给你带来任何问题。

我们来简要浏览一下这些例外情况吧。

3. CLR类型与动态类型之间的转换

你不能将所有CLR类型都转换为object，CLR类型与dynamic之间对于转换的限制与此类似。例外的类型包括指针和System.TypedReference等。由于dynamic在CLR级别就是object，因此这些类型不包括在内也就一点儿也不奇怪了。

你也许还注意到我之前写的是将“dynamic类型的表达式”转换为CLR类型，而不是dynamic类型本身。这个细微的差别有助于类型推断，以及其他需要考虑类型间隐式转换的情况；一般来说，如果两个类型可以进行双向的隐式转换，情况就会变得很糟。考虑转换时基本上限制了这种情况，例如考虑下面的隐式类型数组：

```
dynamic d = 0;
string x = "text";
var array = new[] { d, x };
```

这个array应该推断为什么类型？如果存在dynamic到string的隐式转换，那么它可以为string[]，也可以为dynamic[]，这会产生歧义并导致编译时错误。但由于转换仅存在于动态表达式，因此编译器可以将string转换为dynamic，反之则不行，这样array就为dynamic[]类型。你不必为此担心，除非你试图操作于某个特殊的场景，并且手边放着一本语言规范。

4. 动态表达式并不总是动态地求值

在某些情况下，CLR完全可以使用普通的静态求值路径对表达式进行求值，即使个别子表达式为动态的。例如下面的as操作符：

```
dynamic d = GetValueDynamically();
string x = d as string;
```

此处不会发生任何动态行为——不管d的值是否为字符串引用。由于使用了as操作符，因此不会应用用户定义的转换，这样C#编译器所使用的IL将与变量为object类型时生成的IL完全相同。

5. 动态求值的表达式并不总是动态类型

在某些情况下，编译器并不完全知道如何对表达式进行求值，但却知道结果的确切类型（假设没有抛出异常）。例如，将动态的值作为参数传递到构造函数中：

```
dynamic d = GetValueDynamically();
SomeType x = new SomeType(d);
```

构造函数调用本身应该进行动态求值。在执行时可能要处理多个重载，但结果将总是一个SomeType引用。因此对x的分配将不会发生动态转换。

还有其他一些情况与此类似。例如，对静态类型的数组使用动态数组索引器，所产生的值将只能为数组元素类型。但你不能假设总是会发生所期望的事情。你可能拥有某方法的多个重载，

所有重载都返回相同的静态类型，但该方法调用表达式的类型将仍然为dynamic。

对于没有发生动态求值或没有产生动态值的情况来说，这些已经足够了。让我们回到能够产生动态的情况，来看看C#编译器做了什么，才能让这一切得以发生。

6. 创建调用点和绑定器

你不需要知道编译器为了使用动态表达式而对其进行处理的具体细节，但看看编译后的代码还是相当有益的。如果你由于其他原因而反编译代码，可能不会太在意其中的动态部分看起来是什么样子。对于这种工作，我选择的工具是Reflector（http://mng.bz/pMXJ），如果你想直接阅读IL的话，可以使用ildasm。

我们将只介绍一个示例——如果深入实现细节的话，我肯定可以讲上整整一章，但这里我们只会给出编译器的一些要点。如果对该示例产生了兴趣，你可能会自己进行更多的试验。但要记住的是，只要行为仍然相同，具体的细节是特定于实现的，它们可能会随着版本的更换而改变。

以下是以Snippy的常见方式呈现的代码片段，位于Main方法内：

```
string text = "text to cut";
dynamic startIndex = 2;
string substring = text.Substring(startIndex);
```

很简单吧？但它实际上包含两个动态操作，一个是调用Substring，一个是将结果（编译时为dynamic）动态地（隐式）转换为字符串。代码清单14-17显示了Snippet类[①]的反编译代码。为了节省空间，我省略了类的声明和隐式的无参构造函数，并对代码重新格式化以减少空格。

代码清单14-17　动态代码的编译结果

```
[CompilerGenerated]
private static class <Main>o__SiteContainer0 {        ❶ 存储调用点
  public static CallSite<Func<CallSite, object, string>> <>p__Site1;
  public static CallSite<Func<CallSite, string, object, object>>
       <>p__Site2;
}

private static void Main() {
  string text = "text to cut";
  object startIndex = 2;                               ❷ 创建转换的调用点
  if (<Main>o__SiteContainer0.<>p__Site1 == null) {
    <Main>o__SiteContainer0.<>p__Site1 =
      CallSite<Func<CallSite, object, string>>.Create(
        new CSharpConvertBinder(typeof(string),
          CSharpConversionKind.ImplicitConversion, false));
  }                                                    ❸ 创建获取子字符串的调用点
  if (<Main>o__SiteContainer0.<>p__Site2 == null) {
    <Main>o__SiteContainer0.<>p__Site2 =
      CallSite<Func<CallSite, string, object, object>>.Create(
        new CSharpInvokeMemberBinder(CSharpCallFlags.None,
          "Substring", typeof(Snippet), null,
           new CSharpArgumentInfo[] {
```

① 友情提示，Snippet类由Snippy自动生成。

```
保存文 ❹        new CSharpArgumentInfo(
本类型            CSharpArgumentInfoFlags.UseCompileTimeType, null),
                new CSharpArgumentInfo(
                  CSharpArgumentInfoFlags.None, null) }));
      }
      string substring =                                  ❺ 调用这两个
                                                            动态调用
    <Main>o__SiteContainer0.<>p__Site1.Target.Invoke(
      <Main>o__SiteContainer0.<>p__Site1,
      <Main>o__SiteContainer0.<>p__Site2.Target.Invoke(
        <Main>o__SiteContainer0.<>p__Site2, text, startIndex));
}
```

我不知道你是怎么想的，反正我是非常庆幸不用编写或阅读这样的代码，当然，以学习为目的的情况除外。尽管没有什么新鲜的内容，但为迭代器块、表达式树和匿名函数所生成的代码也是相当可怕的。

内嵌的静态类用于存储该方法所有的调用点❶，因为它们只需要创建一次。（如果每次都要创建，那缓存也就没什么用了！）多线程情况下有可能会多次创建调用点，但这对效率的影响微乎其微，这意味着在实现后期创建（lazy creation）时根本没有使用锁。如果某个调用点的实例被另一个所替换，这并不是什么大问题。使用动态绑定的各个方法都有各自的站点容器：泛型方法肯定是这种情况，因为调用点会因类型参数的不同而变化。别的编译器实现可能会为所有非泛型方法使用一个站点容器，为所有包含单个类型参数的泛型方法使用一个站点容器，等等。

创建完调用点之后（❷和❸），将简单地进行调用。首先调用的是`Substring`（从语句的最内层向外阅读代码），然后对结果进行转换❺。这时我们又将拥有一个静态类型的值，可以将其分配给`substring`变量。

我还想强调代码的另外一个方面：一些静态类型信息保存在调用点中的方式。类型信息本身位于委托的签名中，而委托用于调用点的类型参数（`Func<CallSite,string,object,object>`），`CSharpArgumentInfo`中的标识[①]指明了该类型信息应该用于绑定器❹[②]。（尽管这是方法的目标，却仍然表示为参数。实例方法被视为静态方法，并隐去了第一个参数`this`。）这对于绑定器的行为来说是至关重要的，就好像在执行时重新编译了代码一样。下面我们来看看它为什么如此重要。

14.4.4 更加智能的C#编译器

C# 4能够让你跨越静态和动态的边界，不只是因为一些代码可以静态绑定，一些可以动态绑定，它还能够在一次绑定的过程中，将这两种概念相结合。它能记住调用点中任何需要知道的信息，然后在执行时与动态值的类型进行合并。

14

① 即`CSharpArgumentInfoFlags`枚举。——译者注

② `CSharpArgumentInfoFlags.UseCompileTimeType`枚举值表示在绑定过程中应考虑参数的编译时类型。详细内容可查阅MSDN。——译者注

1. 在执行时保存编译器行为

计算出绑定器行为的理想模式是，假设源代码中没有动态值，我们知道值的确切类型[①]：即执行时实际值的类型 。这仅适用于表达式中的动态值；任何在编译时知道的类型，都仍将用于查找，如成员决策。我将给出两个示例以示区别。

代码清单14-18展示了单个类型中简单的方法重载。

代码清单14-18　单个类型中的动态重载决策

```
static void Execute(dynamic x, string y)
{
   Console.WriteLine("dynamic, string");
}

static void Execute(dynamic x, object y)
{
   Console.WriteLine("dynamic, object");
}
...
object text = "text";
dynamic d = 10;
Execute(d, text);          // 打印"dynamic, object"
```

这里的text是个重要的变量。其编译时类型为object，但执行时的值为字符串引用。对Execute的调用是动态的，因为我们使用了动态变量d作为一个参数，但重载决策使用了text的静态类型，因此结果为dynamic、object。如果text变量也声明为dynamic，则将使用另一个重载[②]。

代码清单14-19与之类似，但这次我们关心的是调用的接收器。

代码清单14-19　类层次结构中的动态重载决策

```
class Base
{
   public void Execute(object x)
   {
      Console.WriteLine("object");
   }
}

class Derived : Base
{
   public void Execute(string x)
   {
      Console.WriteLine("string");
   }
}
...
```

① 实际情况比这要更加复杂——如果实际类型在另一个程序集内部将会如何？例如，我们不会希望通过类型推断将其用于泛型方法的类型参数。绑定器将根据调用上下文和实际类型选择“最佳可访问的类型”。

② 言外之意，如果无法得到参数的编译时类型（尽管编译时text仍然为object，但由于它声明为dynamic，在编译后的代码中，该参数所对应的CSharpArgumentInfoFlags的枚举值为None，即绑定时不附加编译时信息。而如果声明为object，对应的CSharpArgumentInfoFlags枚举值为UseCompileTimeType，即在绑定时使用参数的编译时类型。可参考代码清单14-17），就使用执行时类型（即string）。——译者注

```
Base receiver = new Derived();
dynamic d = "text";
receiver.Execute(d);                    <—— 打印"object"
```

在代码清单14-19中，receiver的执行时类型为Derived，因此你可能会认为将调用Derived中引入的重载。但receiver的编译时类型为Base，因此绑定器将会从备选方法中选择一个可以用来静态绑定的方法[①]。

尽管所有这些决策都将稍后进行[②]，一些编译时检查仍然可用，即使对于那些整个绑定都在执行时进行的代码。[③]

2. 动态代码的编译时错误

正如我在本章开头所说，动态类型的缺点之一，就是将一些通常在编译时可以检测到的错误，推迟到了执行时，进而抛出异常。在很多情况下，编译器只能寄希望于你知道你所做的事情，但能帮的忙，编译器是一定会帮的。

最简单的例子是，调用一个静态类型接收器的方法（即静态方法）时，不论动态值在执行时是什么类型，都无法找到有效的重载。代码清单14-20展示了3个无效的调用，其中两个将被编译器捕获。

代码清单14-20　在编译时捕获动态调用的错误

```
string text = "cut me up";
dynamic guid = Guid.NewGuid();
text.Substring(guid);
text.Substring("x", guid);
text.Substring(guid, guid, guid);
```

我们对string.Substring进行了3次调用。编译器知道确切的重载集合，因为它知道text为静态类型。编译器不会为难第一次调用，因为它无法得知guid的类型——如果为整数，则万事大吉。但最后两行将抛出错误，因为没有以字符串为第一个参数的重载，也没有包含3个参数的重载。编译器能肯定这些调用在执行时会失败，因此以编译时错误取而代之是合情合理的。

略微复杂点的例子是类型推断。如果动态值用来推断泛型方法调用中的类型参数，则在执行前我们无法知道实际的类型参数，并且事先也无法验证。但任何不使用动态值来进行推断的类型参数，都可能在编译时导致类型推断错误。代码清单14-21展示了这样一个示例。

代码清单14-21　混合静态和动态值的泛型类型推断

```
void Execute<T>(T first, T second, string other) where T : struct
{
}
...
dynamic guid = Guid.NewGuid();
Execute(10, 0, guid);
```


① 同样观察编译后的代码，可以发现receiver所对应的CSarpArgumentInfoFlags为UseCompileTime- Type。——译者注

② 即执行时。——译者注

③ 因此可以得出结论：所有在编译时确定的信息（即未使用dynamic声明），都将继续用于执行时。——译者注

```
Execute(10, false, guid);
Execute("hello", "hello", guid);
```

同样，第一个调用可以通过编译，但执行时会失败。第二个调用无法通过编译，因为T不能同时为int和bool，并且它们之间也不存在转换关系。第三个调用也无法通过编译，因为T被推断为string，这违反了必须为值类型的约束。

编译器是保守的：如果它认为某段代码不可能成功，就只会以错误来通知失败，并且它只能执行这方面相对简单的测试。有些情况下，我们可以很明显地（或可证明）看出代码无法工作，但编译器却允许代码通过编译。当然，如果某行特殊的代码永远无法工作，那么执行该行代码的单元测试也将失败，因此如果你的单元测试有很好的代码覆盖率，编译器检查过于简单这个性质就不会带来什么问题。如果它确实发现了问题，可以认为是一种奖赏。

编译器能为我们做的最重要的几个部分，我们都已经做了介绍。但你绝对不能到处使用dynamic。它是有限制的，其中一些很痛苦，但更多的是晦涩。

14.4.5　动态代码的约束

几乎在任何可以使用类型名称编写普通C#代码的地方，你都可以使用dynamic。不过还有一些例外情况。我不会全盘介绍，但会涵盖你最有可能碰到的那些情形。

1. 不能动态处理扩展方法

我们已经看到，编译器在调用点内部生成一些调用的上下文。特别地，调用知道编译器所知道的静态类型。但在当前的C#版本中，它还不知道调用所在的源文件中，using指令到底引入了哪些命名空间。也就是说在执行时它不知道哪些扩展方法是可用的。

这不仅意味着不能调用动态值的扩展方法，还意味着不能将动态值作为参数传入扩展方法。编译器推荐了两种变通方案。如果你知道使用哪个重载，可以在方法内将动态值转换为正确的类型。或者，假设你知道扩展方法所在的静态类型，可以像调用普通的静态方法那样进行调用。代码清单14-22展示了一个失败的调用和这两种解决方法。

代码清单14-22　用动态参数调用扩展方法

```
dynamic size = 5;
var numbers = Enumerable.Range(10, 10);
var error = numbers.Take(size);                    <— 编译时错误
var workaround1 = numbers.Take((int) size);
var workaround2 = Enumerable.Take(numbers, size);
```

如果你想调用动态值的扩展方法，这两种方案也是可行的，尽管使用隐式的this值时，强制转换看上去会很丑陋[①]。

2. 委托与动态类型之间转换的限制

在转换Lambda表达式、匿名方法或方法组时，编译器需要知道委托（或表达式）的确切类型。你不能不加转换就将它们设置为简单的Delegate或object变量。对于dynamic来说也是如

① “使用隐式的this值”是指使用普通的调用扩展方法的形式，但与之前的情形类似，需要进行强制转换，如((IEnumerable<int>)numbers).Take(size)。——译者注

此。强制转换可以满足编译器的要求。如果稍后要动态地执行该委托，那么这样做在某些情况下就非常有用。你还可以将动态类型作为委托的参数。

代码清单14-23展示了一些示例，其中一些可以编译，一些则不行。

代码清单14-23　动态类型和Lambda表达式

```
dynamic badMethodGroup = Console.WriteLine;
dynamic goodMethodGroup = (Action<string>) Console.WriteLine;

dynamic badLambda = y => y + 1;
dynamic goodLambda = (Func<int, int>) (y => y + 1);

dynamic veryDynamic = (Func<dynamic, dynamic>) (d => d.SomeMethod());
```

注意，重载决策的工作方式意味着你不能在不进行强制转换的情况下，对Lambda表达式进行动态地绑定调用——即使被调用的唯一方法可能在编译时具有已知的委托类型。例如，下面的代码无法通过编译：

```
void Method(Action<string> action, string value)
{
   action(value);
}
...
dynamic text = "error";
Method(x => Console.WriteLine(x), text);        <— 编译时错误
```

有必要指出的是，LINQ与`dynamic`的交互不会导致任何损失。你可以拥有一个以`dynamic`为元素类型的强类型集合，你还可以使用扩展方法、Lambda表达式甚至查询表达式。该集合可以包含不同类型的对象，它们在执行时将表现出恰当的行为，如代码清单14-24所示。

代码清单14-24　查询动态元素的集合

```
var list = new List<dynamic> { 50, 5m, 5d };
var query = from number in list
            where number > 4
            select (number / 20) * 10;

foreach (var item in query)
{
   Console.WriteLine(item);
}
```

以上将打印`20`、`2.50`和`2.5`。我故意将元素除以20再乘以10，这样可以展示`decimal`和`double`的区别：`decimal`类型将以非规范化的方式保留精度，因此显示的为`2.50`而非`2.5`。第一个值为整型，因此将使用整型的除法，结果为`20`而非`25`。

3. 构造函数和静态方法

你可以通过指定动态实参的方式来动态调用构造函数和静态方法，但你不能对一个动态类型调用构造函数或静态方法。因为你无法指定具体的类型。

如果你遇到要使用这种动态的情况，可以考虑使用实例方法，例如创建工厂类型。你会发现可以使用简单的多态和接口获得动态行为，而不必使用静态类型。

14

4. 类型声明和泛型类型参数

你不能声明一个基类为dynamic的类型。你同样不能将dynamic用于类型参数的约束，或作为类型所实现的接口的一部分。你可以将其用于基类的类型实参，或在声明变量时将其用于接口。例如，下面的声明是无效的：

- `class BaseTypeOfDynamic:dynamic`
- `class DynamicTypeConstraint<T>where T:dynamic`
- `class DynamicTypeConstraint<T>where T:List<dynamic>`
- `class DynamicInterface:IEnumerable<dynamic>`

而以下声明则有效：

- `class GenericDynamicBaseClass:List<dynamic>`
- `IEnumerable<dynamic> variable;`

大多数有关泛型的约束都是由于dynamic类型并不是真正以.NET类型存在。CLR不了解它——代码中的任何dynamic都将被适当地翻译为应用了DynamicAttribute特性的object。（对于List<dynamic>、Dictionary<string, dynamic>这样的类型来说，该特性指出了类型的哪些部分是动态的。）只有当动态性质需要在元数据中表示时，才会应用DynamicAttribute。局部变量不需要该特性，因为在编译并发现它们的动态性质后，不需要检查它们的任何东西。

所有动态行为，都是通过编译器智能地翻译源代码，以及执行时智能的库[①]来实现的。dynamic和object的这种等价性随处可见，特别是在查看typeof(dynamic)和typeof(object)的时候更为明显，它们返回相同的引用。通常，如果发现dynamic类型不能满足你的要求，记住它在CLR中是什么，并看看这是否能解释该问题。它可能无法提供解决方案，但至少可以提前知道要发生的事情。

这就是我要介绍的关于C# 4 dynamic的全部细节，但如果要对动态类型这个话题有一个全面认识，还应该介绍的一个部分是：动态响应。能够动态地调用代码是一回事，能够动态地响应这些调用是另外一回事。

当然，如果你只是动态地调用第三方代码，甚或使用前面介绍的多重分发，都不需要担心这一点。我明白至少现在，你自认为已经掌握了动态类型，因为我们已经涉及了很多内容。你完全可以跳过下一节，等以后再阅读——本书其他部分都不依赖于此。不过话说回来，它还是很有趣的。

14.5　实现动态行为

C#语言没有为实现动态行为提供任何帮助，而.NET框架则不然。能够动态响应的类型必须实现IDynamicMetaObjectProvider，大多数情况下内嵌的两个实现都能完成大部分工作。我们将研究这两个类型，并介绍一个非常简单的IDynamicMetaObjectProvider实现，以展示所涉及的内容。这三种方法互不相同，我们将从最简单的ExpandoObject开始。

① 指DLR库。——译者注

14.5.1 使用ExpandoObject

System.Dynamic.ExpandoObject乍看上去像个古怪的野兽。它只有一个无参的公共构造函数。除了各个接口的显式实现外，它没有公共方法。比较重要的接口为IDynamicMetaObjectProvider和IDictionary<string,object>。（它实现的其他接口均为IDictionary<string,object>所扩展的接口。）对了，它还是封闭的，所以不能继承它从而实现有用的行为。只有用dynamic引用或实现某个接口时，才能使用ExpandoObject。

1. 设置或获取单独的属性

实现字典接口暗示了它的用途——通过名称存储对象。而这些名称也可用作动态类型的属性。代码清单14-25展示了这两种方式。

代码清单14-25 用ExpandoObject存储和获取值

```
dynamic expando = new ExpandoObject();
IDictionary<string, object> dictionary = expando;
expando.First = "value set dynamically";
Console.WriteLine(dictionary["First"]);

dictionary["Second"] = "value set with dictionary";
Console.WriteLine(expando.Second);
```

代码清单14-25简单地将字符串作为字典的值——实际上你可以在IDictionary<string,object>中使用任何对象。如果将值指定为一个委托，你可以像调用expando的方法那样调用该委托，如代码清单14-26所示。

代码清单14-26 用委托伪造ExpandoObject的方法

```
dynamic expando = new ExpandoObject();
expando.AddOne = (Func<int, int>) (x => x + 1);
Console.Write(expando.AddOne(10));
```

尽管看上去像方法调用，但也可以看成是访问一个返回委托的属性，然后调用该委托。如果你创建了一个静态类型的类，它包含一个类型为Func<int,int>的AddOne属性，调用的语法是完全相同的。C#生成的调用AddOne的代码实际上确实是“调用成员”的操作，而不是访问属性然后再调用，不过ExpandoObject知道该做什么。如果你愿意，完全可以访问属性以获取委托。

让我们介绍一个稍微大一点的示例，尽管我们仍然不会做什么特别复杂的事情。

2. 创建DOM树

我们要用expando创建一颗XML DOM的镜像树。下面的实现相当简陋，只能用于简单的演示，而不能用于真实应用。特别地，它还假设我们不用担心任何XML命名空间。

树中的每个节点都有两个名/值对：XElement——储存用于创建节点的原始LINQ to XML元素，ToXml——储存返回节点XML字符串的委托。你可以仅仅调用node.XElement.ToString()返回字符串，但这次我们要使用ExpandoObject和委托。需要指出的是，我们将使用ToXml来代替ToString，因为设置expando的ToString属性时不能覆盖普通的ToString方法。这会引起

14

混淆，因此我们换了一个名字[①]。

有趣的部分并不是固定的名称，而是它依赖于真正的XML。我们打算完全忽略特性，而所有原始XML中原始元素的子元素，都能通过与该子元素同名的属性来访问。例如，考虑如下所示的XML：

```
<root>
  <branch>
    <leaf />
  </branch>
</root>
```

假定动态变量root表示root元素，我们可以通过两个简单的属性，用一行语句访问叶子节点：

```
dynamic leaf = root.branch.leaf;
```

如果一个元素在其父元素内出现多次，用同名属性访问时只能指向第一个元素。那么要想访问所有的元素，必须使用以元素名加List后缀为名称的属性，它为一个`List<dynamic>`，按文档中的顺序返回所有同名的元素。也就是说，访问仍将表示为`root.branchList[0].leaf`或`root.branchList[0].leafList[0]`。注意，我们在列表中使用了索引器——你不能自行定义expando中索引器的行为。

所有这些内容的实现都非常简单，只需要一个递归方法来处理所有工作，如代码清单14-27所示。

代码清单14-27　使用ExpandoObject实现简化的XML DOM转换

```
public static dynamic CreateDynamicXml(XElement element)
{
   dynamic expando = new ExpandoObject();                    ❶ 设置简单的属性
   expando.XElement = element;
   expando.ToXml = (Func<string>)element.ToString;

   IDictionary<string, object> dictionary = expando;         ❷ 将委托作为属性
   foreach (XElement subElement in element.Elements())
   {
      dynamic subNode = CreateDynamicXml(subElement);
      string name = subElement.Name.LocalName;               ❸ 递归处理子元素
      string listName = name + "List";
      if (dictionary.ContainsKey(name))                      ❹ 将重复的元素添加到列表
      {
         ((List<dynamic>) dictionary[listName]).Add(subNode);
      }
      else
      {                                                      ❺ 新建列表并设置属性
         dictionary[name] = subNode;
         dictionary[listName] = new List<dynamic> { subNode };
      }
   }
   return expando;
}
```

①实际上，如果你使用ToString而不是ToXml，效果是一样的。但这里并没有覆盖（override）原有的ToString方法，而是编写了一个新的方法，如：public new string ToString(){...}。——译者注

如果不处理列表，代码清单14-27还能更简单。我们动态地设置`XElement`和`ToXml`属性（❶和❷），但却不能动态设置元素或其列表，因为我们无法在编译时知道它们的名称[①]。我们使用字典表示来代替（❹和❺），这样可以方便地检查重复的元素。对于某个特殊的键，expando是否包含值，不能仅通过将其作为属性来访问而得知；访问未定义的属性将导致异常。在动态代码中对子元素的递归调用，就像在静态代码中调用一样直接；我们只是对各个子元素递归地调用方法❸，并使用其结果生成适当的属性。

我们需要一些XML作为示例，以图片来生动地展示和以原始格式来展示一样，都会很有帮助。我们将使用表示书籍的简单结构。每本书都包含唯一的名称，用特性来表示，并可能包含多个作者，每个作者为一个元素。图14-4展示了整个文件的树结构，以下文本为原始XML。

```
<books>
  <book name="Mortal Engines">
    <author name="Philip Reeve" />
  </book>
  <book name="The Talisman">
    <author name="Stephen King" />
    <author name="Peter Straub" />
  </book>
  <book name="Rose">
    <author name="Holly Webb" />
    <excerpt>
      Rose was remembering the illustrations from
      Morally Instructive Tales for the Nursery.
    </excerpt>
  </book>
</books>
```

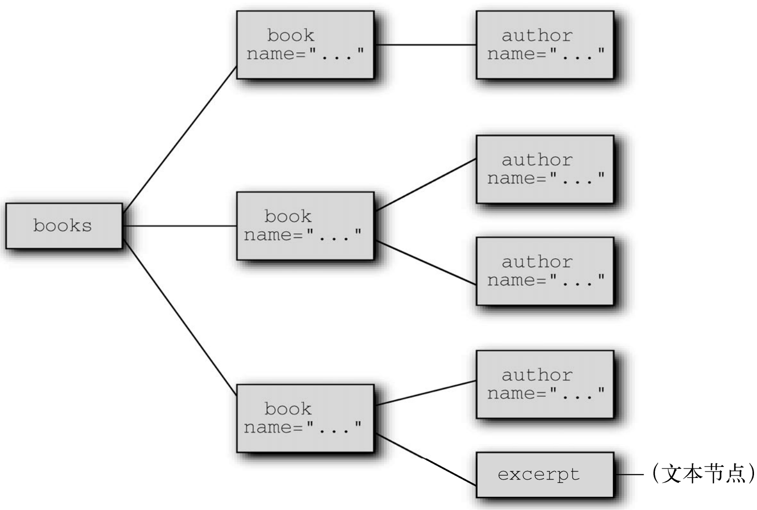


图14-4　XML示例文件的树形结构

① 这颇具讽刺意味——静态的名称可以动态地设置，但动态的名称却只能使用静态类型。

代码清单14-28展示了一个简单的示例，用包含ToXml和XElement属性的expando代码处理XML文档。books.xml文件包含如图所示的XML文档。

代码清单14-28　使用由expando创建的动态DOM

```
XDocument doc = XDocument.Load("books.xml");
dynamic root = CreateDynamicXml(doc.Root);
Console.WriteLine(root.book.author.ToXml());
Console.WriteLine(root.bookList[2].excerpt.XElement.Value);
```

如果你熟悉XElement.Value属性，知道它返回元素内的文本，那么代码清单14-28就再没有什么惊喜之处了。输出结果正如我们所料：

```
<author name="Philip Reeve" />
Rose was remembering the illustrations from
Morally Instructive Tales for the Nursery.
```

这一切都很好，但对于我们使用的DOM也存在一些问题：

- 无法处理特性；
- 每个元素名称都需要两个属性，因为需要表示列表[①]；
- 覆盖ToString()方法要好于添加额外的属性；
- 结果是易变的——以后可以在代码中添加自己的属性[②]；
- 尽管expando是易变的，但它无法反映基础XElement（也是易变的）的任何变化[③]；
- 很可能会发生命名冲突，如节点包含Foo和FooList元素，或元素名为XElement或ToXml；
- 预先生成了整个树，但如果只需要一小部分节点的话，等于做了大量无用功。

要解决这些问题，需要更多的控制，而不仅仅是设置属性。让我们来看看DynamicObject。

14.5.2　使用DynamicObject

DynamicObject与DLR的交互比ExpandoObject要更加强大，且比实现IDynamicMetaObjectProvider要简单得多。尽管它并不是一个真正的抽象类，但只有继承它才能做些有用的事情——而且唯一的构造函数还是受保护的，因此将其视为抽象类可能更加实际。

你可能需要覆盖4类方法：

- TryXXX()调用方法，表示对对象的动态调用；
- GetDynamicMemberNames()，返回可用成员的列表；
- 普通的Equals()、GetHashCode()和ToString()方法仍然可以照常覆盖；
- GetMetaObject()，返回DLR使用的元对象。

---

① 如book和bookList属性。——译者注

② 由于CreateDynamicXml方法返回的是一个ExpandoObject，因此你当然可以像使用普通的ExpandoObject那样添加属性。——译者注

③ 所创建的expando实际上是XML树的一个镜像，但当这个镜像创建好之后，对原XML所做的任何修改都不能反映到这个镜像中。——译者注

我们看一下除最后一条以外的其他方法，以改进XML DOM表示法。我们将在下一节从零开始实现IDynamicMetaObjectProvider的时候，对元对象进行讨论。此外，在派生类型中创建新的对象也很有用，尽管调用者很有可能将实例作为动态值使用。在采取这些步骤之前，我们需要一个类来保存所有这些成员。

1. 准备工作

由于我们要继承DynamicObject，而不仅仅是调用它的方法，因此需要先声明一个类。代码清单14-29展示了一个基本骨架，稍后我们会让它变得血肉丰满。

代码清单14-29 DynamicXElement的骨架

```
public class DynamicXElement : DynamicObject
{
   private readonly XElement element;          ❶ 该实例所包装的XElement

   private DynamicXElement(XElement element)   ❷ 私有的构造函数，防止直接实例化
   {
      this.element = element;
   }

   public static dynamic CreateInstance(XElement element)   ❸ 创建实例的公共方法
   {
      return new DynamicXElement(element);
   }
}
```

DynamicXElement类仅仅包装了一个XElement❶。这将是我们所拥有的全部状态，是它自身内部的一个非常重要的设计决策。在之前创建的ExpandoObject内，我们对XML的结构进行递归，并生成整个树的镜像。这么做是必需的，因为我们无法在自定义代码中拦截属性访问。这种开销显然比DynamicXElement要大，因为后者只有在真正需要的时候才会对树中的元素进行包装。此外，那样做还意味着创建expando之后，对XElement所做的所有修改实际上都将丢失；例如，如果添加了其他子元素，将不会作为expando的属性出现，因为在生成快照时这些子元素还不存在。轻量级的包装方式将总是“实时的”——对树所做的任何更改都将通过包装器展现出来。

这么做的缺点是，无法提供与之前相同的识别方式。使用expando，两次对表达式root.book.author求值将返回相同的引用。而使用DynamicXElement，每次求值时都将创建新的实例，以包装子元素。我们可以实现某种智能缓存来规避这一点，但它很快就会变得异常复杂。

代码清单14-29中，DynamicXElement的构造函数为私有的❷，并提供一个公共的静态方法来创建实例❸。方法的返回值为dynamic，这正是我所期望的开发者对该类的使用方式。还有一种方法是创建一个单独的公共静态类，然后创建XElement的扩展方法，并在方法内部保留DynamicXElement。该类本身就是一个实现的详细说明；除非想动态地使用该类，否则没有多大意义。

骨架创建完成之后我们就可以开始添加特性了。从最简单的东西开始：向普通类添加方法和索引器。

2. **DynamicObject**支持的简单成员

在探讨expando时，我们总是会添加两个成员：ToXml方法和XElement属性。然而这次我们不再需要用新的方法将对象转换为字符串的表示形式；我们可以覆盖普通的ToString()方法。我们还可以提供XElement属性，就像是在写其他类一样。

更棒的是，如果某些行为不需要真正的动态特性，你大可不必将它们实现为动态的。在相关的元对象使用任何TryXXX方法之前，它都会检查该成员是否已经作为简单的CLR成员而存在。如果是，将调用这个CLR成员。这样就变得简单多了。

DynamicElement还将拥有两个索引器，分别用来访问特性和替代元素列表。代码清单14-30展示了要添加到该类中的新代码。

代码清单14-30　向DynamicXElement中添加非动态成员

```
public override string ToString()                    ❶ 正常地覆盖ToString()
{
   return element.ToString();
}

public XElement XElement                             ❷ 返回包装后的元素
{
   get { return element; }
}

public XAttribute this[XName name]                   ❸ 获取特性的索引器
{
   get { return element.Attribute(name); }
}

public dynamic this[int index]                       ❹ 获取兄弟元素的索引器
{
   get
   {
      XElement parent = element.Parent;
      if (parent == null)                            ❺ 是否为根元素
      {
         if (index != 0)
         {
             throw new ArgumentOutOfRangeException();
         }
         return this;
      }
      XElement sibling = parent.Elements(element.Name)    ❻ 找到适当的兄弟元素
                               .ElementAt(index);
      return element == sibling ? this
                                : new DynamicXElement(sibling);
   }
}
```

代码清单14-30中的代码不少，但大多数都很简单。我们通过代理调用XElement，覆盖了ToString()❶。如果要实现值的相等性，可以用同样的方法覆盖Equals()和GetHashCode()。返回基础元素的属性❷和返回特性的索引器❸同样也很简单，但值得注意的是，特性索引器的参

数只能使用XName；如果执行时提供了一个字符串，DynamicObject将为我们调用隐式转换，将其转换为XName。

这段代码最难的部分是理解以int为参数的索引器❹的功能。从用法方面来解释可能会简单一些，即让一个元素既表示单个元素，又表示同名的子元素列表，以这种方式来避免使用额外的列表属性。图14-5展示了前一个示例XML，用一些表达式来访问不同的节点。

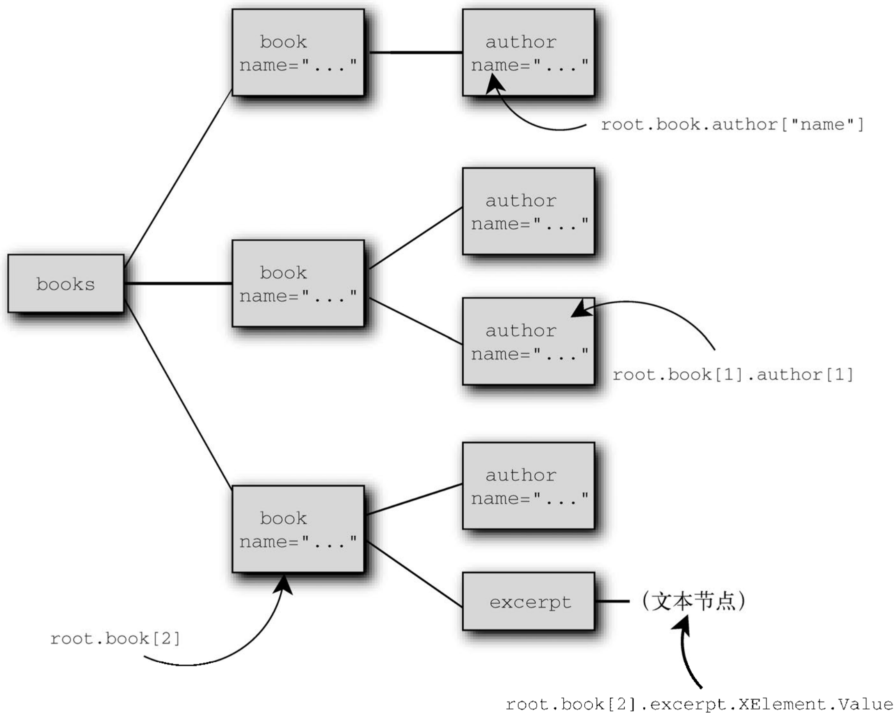


图14-5　使用DynamicXElement选择数据

只要理解了索引器的功能，实现起来就相当简单了，唯一复杂的情况是我们可能已经位于树的顶部了❺。否则只需要获取该元素的所有兄弟元素，然后选择我们想要的那个❻。

目前为止，除了CreateInstance()的返回类型外，没有使用任何动态特性——所有这些示例都没法运行，因为还没有编写获取子元素的代码。现在就来修补这一点。

3. 覆盖**TryXXX**方法

在DynamicObject中，我们通过覆盖TryXXX方法来响应动态调用。这组方法共12个，分别代表不同的操作类型，如表14-1所示。

14

表14-1　**DynamicObject**中的**TryXXX**虚方法

名　　称	所表示的调用类型（**x**为动态对象）
TryBinaryOperation	二元运算，如x + y
TryConvert	转换，如(Target)x
TryCreateInstance	对象创建表达式：C#中没有等价的表达式
TryDeleteIndex	移除索引的操作：C#中没有等价的操作
TryDeleteMember	移除属性的操作：C#中没有等价的操作
TryGetIndex	获取索引器中的项，如x[10]
TryGetMember	获取属性的值，如x.Property
TryInvoke	将x视为委托直接调用，如x(10)
TryInvokeMember	成员调用，如x.Method()
TrySetIndex	设置索引器中的项，如x[10] = 20
TrySetMember	设置属性，如x.Property = 10
TryUnaryOperation	一元运算，如!x或-x

每个方法的返回类型均为布尔型，指明该绑定是否成功。每个方法都将适当的绑定器作为第一个参数，并且如果该操作逻辑上包含参数（如方法的参数，或索引器的索引），这些参数都将表示为object[]。最后，如果该操作有返回值（除了设置和删除操作外都可能有返回值），将会有一个object类型的out参数来获取该值。

操作决定了绑定器的实际类型，不同的操作对应不同的绑定器类型。例如，TryInvokeMember的完整签名为：

```
public virtual bool TryInvokeMember(InvokeMemberBinder binder,
   object[] args, out object result)
```

你只需要覆盖你所支持的动态操作的方法。在本例中，我们拥有一个只读的动态属性（用来获取元素），因此需要覆盖TryGetMember()，如代码清单14-31所示。

代码清单14-31　使用TryGetMember()实现动态属性

```
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
   XElement subElement = element.Element(binder.Name);
   if (subElement != null)
   {
      result = new DynamicXElement(subElement);
      return true;
   }
   return base.TryGetMember(binder, out result);
}
```

❶ 查找第一个匹配的子元素

❷ 如果找到了，则新建动态元素

❸ 否则使用基类的实现

代码清单14-31的实现非常简单。绑定器包含被请求的属性的名称，因此可以在树中查找适合的子元素❶。如果存在，就用它新建一个DynamicXElement，赋值给输出参数result，并返

回true，表明对该调用的绑定成功了❷。如果不存在具有该名称的子元素，就调用基类的TryGetMember()实现❸。所有TryXXX方法的基类实现都返回false，如果存在输出参数，则设置为null。我们也可以显式地这么做，但那样就会包含两条语句：一条设置输出参数，一条返回false。如果你更喜欢长一点的语句，我也没理由阻止你这么做——基实现在完成了所有必要的工作来表明绑定失败方面，确实方便了一点儿。

我避开了一个复杂之处：绑定器还有一个属性（IgnoreCase），可以指明所绑定的属性是否不区分大小写。例如，Visual Basic不区分大小写，因此其绑定器对于该属性就返回true，而C#则返回false。此种情况略显棘手[①]。以不区分大小写的方式查找元素，TryGetMember需要做更多的工作（“更多的工作”总是不爽的，但这并不是我们逃避工作的理由），而且在使用（数字）索引器选择兄弟元素时，也会有很多问题。对象是否应该记住区分大小写与否，并在以后选择兄弟元素时采用相同方式？那样的话，你会很容易陷入行为无法预测和描述的窘境[②]。这种阻抗不匹配（impedance mismatch）[③]也会出现在其他类似情况下[④]。如果你追求完美，有可能会作茧自缚。相反，应该用肯定能实现和维护的实用方案，然后列出约束。

一切就绪后，可以用代码清单14-32来测试DynamicXElement。

代码清单14-32 测试DynamicXElement

```
XDocument doc = XDocument.Load("books.xml");
dynamic root = DynamicXElement.CreateInstance(doc.Root);
Console.WriteLine(root.book[2]["name"]);
Console.WriteLine(root.book[1].author[1]);
Console.WriteLine(root.book);
```

当然，这个类还可以更复杂。我们可以添加Parent属性，对树进行向上导航，或修改代码，以使用方法访问子元素，并以访问属性的方式表示特性。原则是完全一致的：那些可以事先知道名称的元素、特性等，都可以实现为普通的类成员。如果想让它变为动态的，就覆盖DynamicObject中的适当方法。

我们最后还要向DynamicXElement添加一个十分优雅的部分，现在就开始吧。

4. 覆盖GetDynamicMemberNames

像Python这类语言，都允许对象列出它所知道的所有名称。例如，在Python中可以使用dir函数输出这样一个列表。这常用于REPL环境，并且在IDE中调试时也十分方便。DLR通过DynamicObject和DynamicMetaObject（稍后介绍）的GetDynamicMemberNames()方法来表示这些信息。我们所要做的，就是覆盖该方法，以提供动态成员名称的序列，这样就可以将对象的属性公布于众了。代码清单14-33展示了DynamicXElement的实现。

① 因为XML是大小写敏感的，因此如果把IngoreCase设置为true，将会带来很多问题。——译者注

② 如果在选择兄弟元素时也忽略大小写，那么book和Book将被认为是兄弟元素，而它们在XML中却不是。——译者注

③ 阻抗不匹配通常用于描述面向对象的应用在向关系型数据库中存放数据时，所遇到的数据表述不一致的问题，详见http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch。——译者注

④ 比如在访问特性时，name和Name将被认为是相同的特性，从而导致错误。——译者注

代码清单14-33　实现DynamicXElement中的GetDynamicMemberNames

```
public override IEnumerable<string> GetDynamicMemberNames()
{
    return element.Elements()
                  .Select(x => x.Name.LocalName)
                  .Distinct()
                  .OrderBy(x => x);
}
```

如你所见，我们需要的只是一个简单的LINQ查询。情况并非总是如此，但我认为很多动态实现都可以像这样使用LINQ。

在本例中，如果多个元素拥有相同的名称，将只返回一次值，并且为了保持一致性，还将对结果进行排序。在Visual Studio 2010的调试器中，可以展开动态对象的Dynamic View，查看属性的名称和值，如图14-6所示。

你可以深入动态对象，逐级显示Dynamic View。在图14-6中，我从文档进入到第一本书，再到作者。作者的Dynamic View显示没有更深层次的信息了。

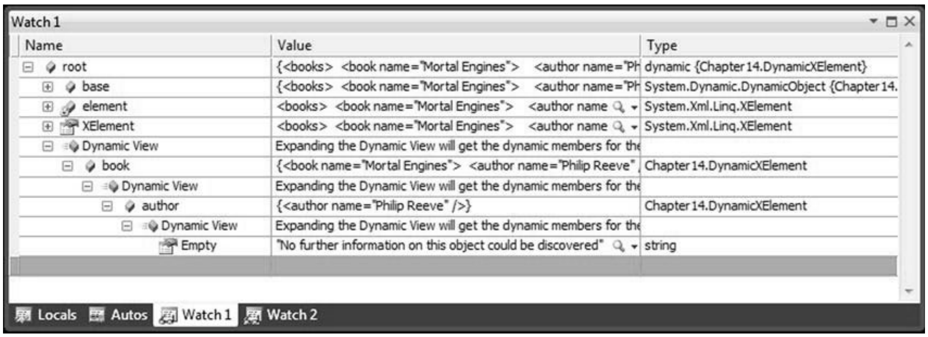


图14-6　在Visual Studio 2010中显示DynamicXElement的动态属性

本书就对DynamicXElement介绍这么多。我觉得DynamicObject是控制性和简单性之间的黄金分割点：它相当简单，而且比ExpandoObject的限制要少得多。如果你需要完全地控制绑定，可以直接实现IDynamicMetaObjectProvider。

14.5.3　实现IDynamicMetaObjectProvider

我并不想涉及过多的细节，但至少要展示一个低级别的动态行为示例。实现IDynamicMetaObjectProvider的难点并不是接口本身，而是构建该接口的唯一方法所返回的DynamicMetaObject。DynamicMetaObject有点像DynamicObject，它包含很多方法，我们可以覆盖它们，并影响相应的行为。比如前面我们覆盖了DynamicObject.TryGetMember，这里我们可以覆盖DynamicMetaObject.BindGetMember。但在被覆盖的方法内，它不会直接处理所需的行为，而会构建一个表达式树来描述行为以及行为产生的环境。这种额外的间接层就是它称为元对象的原因。

我将直接介绍一个示例，然后再进行简短的解释。我真希望能讲解清楚交互级之间的区别，这就像是在研究JIT编译器的实现。大多数C#开发者没有必要了解这些细节，除非你要编写能够动态响应的库，并且对性能有一定的要求，或者你要构建自己的动态语言。如果真是这样，那么恭喜你，去找一个比这个小示例更全面的资源吧。

我们的示例并没有多么智能，它是一个Rumpelstiltskin类型。我们要通过一个给定的名称（存储在一个非常常见的字符串变量里）来创建Rumpelstiltskin的实例，然后在该对象上调用方法，直到调用方法的名称与字符串相同。该对象将根据我们的猜测输出恰当的响应结果[①]。具体来说，代码清单14-34展示了我们最终要运行的代码。

代码清单14-34　最终目标：动态地调用方法，直到命中正确的名称

```
dynamic x = new Rumpelstiltskin("Hermione");
x.Harry();
x.Ron();
x.Hermione();
```

对象并不叫做Rumpelstiltskin（侏儒怪），那样太明显了。相反，我们会使用其他魔法师的名字[②]，尽管他们都不以炼金术见长[③]。我们的目的是对前两次调用予以否认，而对第三次则甘拜下风。我们的方法调用还将返回布尔值，来表明猜测是否正确，但简便起见，这里并不使用这个结果。

首先看一下Rumpelstiltskin类型。记住这可不是元对象，我们一会儿才会讲到，代码清单14-35显示了完整的代码。

代码清单14-35　不含元对象代码的Rumplelstiltskin类型

```
public sealed class Rumpelstiltskin : IDynamicMetaObjectProvider
{
   private readonly string name;
   public Rumpelstiltskin(string name)              ❶ 构造新的实例
   {
      this.name = name;
   }

   public DynamicMetaObject GetMetaObject(Expression expression)
   {
      return new MetaRumpelstiltskin(expression, this);     ❷ 公开动态行为
   }

   private object RespondToWrongGuess(string guess)
   {                                                ❸ 响应猜测
      Console.WriteLine("No, I'm not {0}! (I'm {1}.)",
         guess, name);
      return false;
```

① 如果你对Rumpelstiltskin（侏儒怪）的童话故事不是很了解，可以浏览维基百科上的文章（http://en.wikipedia.org/wiki/Rumpelstiltskin）。这样你才能明白这个例子。

② 很显然，这些魔法师是《哈利·波特》系列的三个主人公：哈利、罗恩和赫敏。——译者注

③ 侏儒怪都擅长炼金术。——译者注

```
    }

    private object RespondToRightGuess()
    {
        Console.WriteLine("Curses! Foiled again!");
        return true;
    }
}
```

该类包含了3个部分。极其普通的构造函数❶，IDynamicMetaObjectProvider仅有方法的实现❷，以及要执行真正工作的两个方法❸。

元对象的构造❷需要知道它所响应的实例，以及用来在调用代码中引用该实例的表达式树。我们将表达式树作为参数，对于自身实例的引用可以使用this，将这两者传递给（元对象的）构造函数。

说明　这两个方法为什么返回object？　你可能会奇怪，为什么这两个方法返回object而不是bool呢？实际上，我一开始实现的是void方法，但不幸的是动态方法调用必须有返回值，并且就我的经验来看，绑定器总是希望能返回object。（可以检查ReturnType属性。）在执行时调用void方法将抛出异常，bool方法也是如此；我们需要自行装箱，使类型能够匹配。我们可以将装箱构建在表达式中，但修改方法的返回类型要简单得多。如果要在现实生活中实现IDynamicMetaObjectProvider，这些就是你要面对的细微之处。

严格地说，我们并不需要这两个响应方法。当构建行为来响应传入的方法调用时，我们可以直接将逻辑表示在表达式树中。但这相对来说比仅仅返回一个调用正确方法的表达式树要复杂。更重要的是，尽管在本例中并不很难，但在其他情况下可能会很糟糕。我们实际上是要在静态世界和动态世界之间搭建一座桥梁，通过适当的参数，将对动态方法调用的响应重定向到静态方法。这使元对象中的代码变得简单。

说到这儿，我们最后来看看代码清单14-36中MetaRumpelstiltskin的代码，它是Rumpelstiltskin的一个私有内嵌类。

代码清单14-36　Rumpelstiltskin动态性的实质——元对象

```
private class MetaRumpelstiltskin : DynamicMetaObject
{
    private static readonly MethodInfo RightGuessMethod =
        typeof(Rumpelstiltskin).GetMethod("RespondToRightGuess",
        BindingFlags.Instance | BindingFlags.NonPublic);

    private static readonly MethodInfo WrongGuessMethod =
        typeof(Rumpelstiltskin).GetMethod("RespondToWrongGuess",
        BindingFlags.Instance | BindingFlags.NonPublic);

    internal MetaRumpelstiltskin
        (Expression expression, Rumpelstiltskin creator)
        : base(expression, BindingRestrictions.Empty, creator)
```

❶ 通过反射得到方法

❷ 将构造委托给基类

```
    {}

    public override DynamicMetaObject BindInvokeMember          ❸ 响应成员调用
       (InvokeMemberBinder binder, DynamicMetaObject[] args)
    {                                                            ❹ 记住真正的对象
       Rumpelstiltskin targetObject = (Rumpelstiltskin)base.Value;
       Expression self = Expression.Convert(base.Expression,
            typeof(Rumpelstiltskin));

       Expression targetBehavior;                                ❺ 确定适当的行为
       if (binder.Name == targetObject.name)
       {
          targetBehavior = Expression.Call(self, RightGuessMethod);
       }
       else
       {
          targetBehavior = Expression.Call(self, WrongGuessMethod,
             Expression.Constant(binder.Name));
       }

       var restrictions = BindingRestrictions.GetInstanceRestriction   ❻ 响应行为和约束
            (self, targetObject);
       return new DynamicMetaObject(targetBehavior, restrictions);
    }
}
```

在我敲下这段代码时，可以感觉得到你看到它之后呆滞的目光。如此密集的代码，看上去有点小题大作。要记住的是，你并不是必须这么做，因此放轻松，让自己完全沉浸在代码的海洋中吧。

代码的前半部分十分简单。我们将那两个响应方法的MethodInfo存储在静态变量中❶（它们不会因实例的不同而改变），然后声明一个构造函数，它不进行任何处理，只是将参数向上传递给基类的构造函数❷。所有真正的工作都位于BindInvokeMember内❸，它解决了两个问题，即对象如何响应方法调用，以及调用哪个响应方法。

我们将判断方法调用的名称和对象的名称[①]是否相同，来决定是调用ResponseToRightGuess还是RespondToWrongGuess进行响应。元对象知道真正的（Rumpelstiltskin对象）实例，因为我们将其传递给了构造函数。我们将使用Value属性再次访问该实例，并将其存储在targetObject变量中❹。我们还需要原本用于创建元对象的表达式树，这样就可以将适当的方法调用绑定到表达式树中。Expression. Convert方法所创建的表达式树与前一行代码中的强制转换是完全等价的。

一旦我们得到了真正的对象，就可以检查它的名称是否与通过InvokeMemberBinder.Name得到的方法调用的名称相同。如果猜测错误，就将适当的方法名称作为参数，传递给Expression.Call，用来构建对该方法的调用❺。我要再次强调的是，此时我们实际上并没有调用这个方法——我们只是在描述这个方法的调用。

① 即Rumpelstiltskin对象的name私有只读字段。——译者注

本例的约束十分简单：如果用相同的参数调用，那么该调用总是以相同的方式进行绑定。但如果在不同的对象上调用，绑定的方式也将不同，因为它们可能具有不同的名称。GetInstanceRestriction返回一个适当的约束，如果我们不管方法调用所在的实例是否相同，而总是希望产生相同的行为，可以使用GetTypeRestriction，它表明对于任何Rumpelstiltskin的实例，都以相同的方式处理调用。完整的源代码中包含了另一种实现[①]，它总是传递实际的方法名称，并将条件判断放入普通方法的内部。

最后，我们新建了一个表示绑定结果的DynamicMetaObject❻。处理绑定的方法所返回的类型与该方法所属的类型相同，这很令人费解，但这就是DLR的工作方式。

这样，我们就完成了全部工作——十指交叉，运行代码，祈祷它能顺利完成吧。如果你像我一样愚笨，可能需要调试好几次才能找到错误之处。正如我所说过的，这并不是所有开发者都必须掌握的内容——它有点像LINQ，使用LINQ的人远比自行实现基于IQueryable的LINQ提供器的人多。剥去其神秘的外衣，一窥究竟，是十分有意义的。但大多数时候，你只需坐享DLR团队的卓越贡献。

14.6　小结

我们似乎远离了C#静态类型这个主流，介绍了动态类型的一些用武之地、C# 4如何实现（前台代码和后台原理两方面），以及如何动态地响应调用。我们顺便提及了COM、Python和反射的一点儿内容，并学习了DLR相关的一些知识。

这并不是关于DLR的完整指南，甚至不是C#对DLR的操作手册。它只是一个包含了很多死角的深入讨论。你通常不会遇到那些晦涩难懂的问题——大多数开发者连最简单的情景也不会频繁使用。我确信如果要介绍DLR，那需要整本书的篇幅，但我希望本书已经给出了足够的细节，使99%的C#开发者不再需要额外的学习就能完成他们的工作。如果你还想了解更多内容，DLR网站上的文档是个不错的切入点（参见http://mng.bz/0M6A）。

如果你用不到dynamic类型，则几乎可以完全忽略动态类型。我建议你在大多数代码中都不要使用——尤其不要用它来逃避创建合适的接口和基类等。在确实需要动态类型的地方，我也会尽可能少地使用。“我在该方法中使用了dynamic，因此我可以将所有东西都变为动态的”，这种态度要不得。

我并不想以过于消极的态度看待dynamic。如果你发现在某个场景中动态类型十分有帮助，你肯定会感谢C# 4的。尽管你可能永远不会在生产代码中使用动态类型，我还是鼓励你试着去玩一玩——我已经被它强烈地吸引住了。你还会发现即使不使用动态类型，DLR也会很有用；本章的Python示例大部分都没有使用任何动态类型的特性，但却使用了DLR来执行包含配置数据的Python脚本。

本章和第13章涵盖了C# 4中所有新增的语言特性。接下来，C# 5中对动态类型的关注点要比C# 4中的更为具体，几乎都是在探讨异步编程。

① 即源代码中的Chapter14.Rumpelstiltskin2类。——译者注

Part 5 第五部分

C# 5：简化的异步编程

要描述C# 5是非常容易的：它包含一个大特性（异步函数）和两个小特性。

第15章全部是关于异步的内容。异步函数特性（通常简称为async/await）的目标是简化异步编程，至少要比之前更简化。它并不是要移除异步编程固有的（inherent）复杂性。你仍然需要考虑按乱序完成的操作结果，或在第一个操作完成之前，用户按下另一个按钮时应如何处理。但它移除了附带的（incidental）复杂性。它能让你既见树木又见森林，并为这些附带的复杂性构建出健壮、可读的解决方案。

以前，异步代码看上去就像意大利面条，当一个异步调用结束并开始调用另一个时，逻辑执行路径将从一个方法跳到另一个方法。有了异步函数，代码看上去就像是同步的，使用熟悉的控制结构（如循环和try/catch/finally块），只不过用一个新的关键字（await）来触发异步执行流。在我看来，这两种方案的可读性是天壤之别。我们将深入介绍这一话题，不仅是语言方面的行为，还包括微软C#编译器的实现。

第16章涵盖了另外两个特性：在第5章看到的foreach迭代的行为有了微小的改变，另外针对C# 4引入的可选参数，还增加了一些特性，可以让编译器自动提供一段代码的行数、成员名和源文件。最后是结束语，我将用我习惯的方式来结束本书。

你可能会误以为内容不够丰富，特别是第16章介绍的特性我还刻意地轻描淡写。不要被迷惑了。异步函数绝对是大手笔，特别是在使用WinRT编写Windows Store应用的时候。WinRT暴露的API都是围绕异步构建的，以解决失去响应的用户界面问题。如果没有异步函数，这些API将会很难使用。有了C# 5的特性，你仍然需要思考，但代码是我能想象到的最清晰的异步代码。因此，与其继续描写它有多么不得了，不如现在就来看看这个特性……

第 15 章

使用async/await进行异步编程

本章内容

- 异步的基本目标
- 编写异步方法和委托
- 编译器对异步的转换
- 基于任务的异步模式
- WinRT中的异步

多年来，异步编程对开发者来说就是一根芒刺。我们都知道，异步编程可在等待某个任务完成时，避免线程的占用，但要想正确地实现编程，仍然十分伤脑筋。

在（相对于其宏伟蓝图来说仍然十分年轻的）.NET Framework中，有三种不同的模型来简化异步编程。

- .NET 1.x中的`BeginFoo/EndFoo`方法，使用`IAsyncResult`和`AsyncCallback`来传播结果。
- .NET 2.0中基于事件的异步模式，使用`BackgroundWorker`和`WebClient`实现。
- .NET 4引入并由.NET 4.5扩展的任务并行库（TPL）。

尽管TPL经过了精心设计，但用它编写健壮可读的异步代码仍然十分困难。虽然支持并行是一个壮举，但对于异步编程的某些方面来说，最好是从语言层面进行修补，而不是纯粹的库。

> 说明　**async/await会改变你的世界观**　本章概要中的话题可能会显得索然无味。它是一个正确的概要，但却没能准确传达我对该特性的激动之情。我使用async/await已经两年了，但仍然觉得像刚入学的小学生。我坚信async/await之于异步编程，一定会像C# 3诞生时LINQ之于数据处理一般，除了处理异步要复杂得多。为达到最佳效果，请以万分激动的声音大声阅读本章。希望我能以我的热情影响你对该特性的看法。

C# 5的这个主要特性基于TPL，因此可以在适用于异步的地方编写同步形式的代码。意大利面式的回调、事件订阅和支离破碎的错误处理都消失不见，取而代之的是能够清晰表达其意图的

异步代码，并且是基于开发者熟悉的结构。它包含一个新的语言构造，可以“等待”（await）一个异步的操作。这个“等待”看上去非常像一个普通的阻塞调用，剩余的代码在异步操作完成前不会继续执行，但实际上它并没有阻塞当前执行线程。如果觉得该语句完全自相矛盾，不要担心，完成本章的学习后，一切就会清晰了。

.NET Framework从版本4.5起全心投入了异步编程的怀抱，并为大量操作提供了异步版本。它们遵循基于任务的新异步模式，横跨多个API提供一致的体验。此外，用于在Windows 8中创建Windows Store应用的全新Windows Runtime平台①，还强制所有长时间运行（或有可能长时间运行）的操作都必须使用异步。总而言之，未来是异步的，在处理额外复杂性的时候不使用这个新的语言特性是愚蠢的。即便你使用的不是.NET 4.5，微软还创建了一个NuGet包（`Microsoft.Bcl.Async`），可以在面向.NET4、Silverlight 4/5、Windows Phone 7.5/8时使用该新特性。

需要清楚的是，C#并不是无所不知的，它无法猜出哪里应该并行或异步地执行操作。编译器是聪明的，但它无法移除异步执行的固有复杂性。你仍然需要仔细思考，但C# 5的魅力在于，所有以前必需的繁琐混乱的样板代码全部消失殆尽。没有了那些让你分心的繁文缛节，你可以专注于那些困难的部分。

请注意，该话题是相当有深度的。其重要性无法估量（说真的，入门级开发者要想彻底弄懂需要几年的时间），但初学时又是十分困难的。与本书其他部分一样，我不会回避复杂性，我们将通过大量的细节一探究竟。

我很可能会让你头痛欲裂，希望你很快能恢复正常。如果你觉得自己要发疯了，不要担心，不只你一个人这么想。迷茫是相当正常的反应。好消息是，在使用C# 5时，这一切都相当简单。只有在想知道背后原理时，才会感到复杂。当然，我们一会儿会这么做的。但在此之前，先来看看如何使用该特性。

我们开始吧。

15.1 异步函数简介

我说了C# 5可以简化异步编程，但却只给出了简短的描述。我们先解决这一点，然后再来看一个例子。

C# 5引入了异步函数（asynchrnous function）的概念。通常是指用`async`修饰符声明的，可包含`await`表达式的方法或匿名函数②。从语言的视角来看，这些`await`表达式正是有意思的地方：如果表达式等待的值还不可用，那么异步函数将立即返回；当该值可用时，异步函数将（在适当的线程上）回到离开的地方继续执行。此前“在这条语句完成之前不要执行下一条语句”的流程依然不变，只是不再阻塞。

稍后我会将含糊的描述分解成更加具体的术语和行为。但现在更需要一个示例，才能对其有所感悟。

15

① 通常称为WinRT。不能将它与Windows RT混淆，后者是指在ARM处理器上运行的Windows 8版本。

② 匿名函数为Lambda表达式或匿名方法。

15.1.1　初识异步类型

我们先来看一个非常简单，但却能实际演示异步编程的例子。我们经常诟病网络延迟会让应用程序反应迟钝，但恰恰是这种延迟证明了异步编程是多么重要。看看代码清单15-1。

代码清单15-1　异步地显示页面长度

```
class AsyncForm : Form
{
   Label label;
   Button button;

   public AsyncForm()
   {
      label = new Label { Location = new Point(10, 20),
                          Text = "Length" };

      button = new Button { Location = new Point(10, 50),
                            Text = "Click" };
      button.Click += DisplayWebSiteLength;          ❶ 包装事件处理程序
      AutoSize = true;
      Controls.Add(label);
      Controls.Add(button);
   }

   async void DisplayWebSiteLength(object sender, EventArgs e)
   {
      label.Text = "Fetching...";
      using (HttpClient client = new HttpClient())
      {
         string text =                                ❷ 开始获取页面
               await client.GetStringAsync("http://csharpindepth.com");
         label.Text = text.Length.ToString();         ❸ 更新UI
      }
   }
}
...
Application.Run(new AsyncForm());
```

代码清单15-1的第一部分简单地创建了UI，并直接为按钮绑定了事件处理程序❶。有趣的是`DisplayWebSiteLength`方法。按下按钮，将获取本书主页的文本❷，并在便签中显示HTML的字符长度❸。不管操作是否成功，`HttpClient`都会恰当地进行释放。一切都如此简单，以至于你可能都忘了如何用C# 4写类似的异步代码。

说明　**释放任务**　我在使用完`HttpClient`之后小心地进行了释放，但却没有释放`GetStringAsync`返回的任务，尽管`Task`也实现了`IDisposable`。幸好，一般来说你并不需要释放任务。其背景有些复杂，Stephen Toub专门就这一话题写了一篇文章进行阐述，文章网址为http://mng.bz/E6L3。

我本可以写一个更小的示例（控制台应用），不过还是觉得代码清单15-1更有说服力。如果移除async和await上下文关键字，将HttpClient替换为WebClient，将GetStringAsync改成DownloadString，代码仍能编译并工作。但是在获取页面内容时，UI将无法响应①。而运行异步版本时（理想情况下通过较慢的网速进行连接），UI仍然能够响应，在获取网站页面时，仍然能够移动窗体。

大多数开发者都知道在开发Windows Form时，有两条关于线程的金科玉律。

- 不要在UI线程上执行任何耗时的操作。
- 不要在除了UI线程之外的其他线程上访问UI控件。

说起来容易，恪守很难。作为练习，你可以尝试一下用不同的方式（但不用C# 5的新特性），来编写类似代码清单15-1的代码。对于这个极其简单的示例来说，使用基于事件的WebClient.DownloadStringAsync方法也不是不可以，但如果加入更多复杂的流控制（错误处理、等待多个页面等），"遗留"代码将很快变得难以维护，而C# 5的代码则可以很自然地进行修改。

现在，你可能觉得DisplayWebSiteLength方法有些神奇：你知道它做了该做的事，但却不知道它是如何做的。我们先中断这个话题，一会儿再来探讨细节。

15.1.2　分解第一个示例

我们先对该方法进行一点扩充，将HttpClient.GetStringAsync从await表达式中分离出来，以强调所涉及的类型：

```
async void DisplayWebSiteLength(object sender, EventArgs e)
{
   label.Text = "Fetching...";
   using (HttpClient client = new HttpClient())
   {
      Task<string> task =
         client.GetStringAsync("http://csharpindepth.com");
      string text = await task;
      label.Text = text.Length.ToString();
   }
}
```

注意，task的类型是Task<string>，而await task表达式的类型是string。也就是说，await表达式执行的是"拆包"（unwrap）操作，至少在被等待的值为Task<TResult>时是这样。（还可以等待其他类型，但Task<TResult>是一个不错的起点。）这是await的一个方面，看上去跟异步编程没什么直接关系，但却让生活更加轻松。

await的主要目的是在等待耗时操作完成时避免阻塞。它是如何在具体线程中工作的呢？我们在方法的开始和结束处都设置了label.Text，所以可以很自然地认为这两条语句都在UI线程执行。但显然我们在等待页面下载的时候没有阻塞UI线程。

① HttpClient在某种程度上是"经过改进的全新"WebClient。它是.NET 4.5之后的首选HTTP API，并且只包含异步操作。如果要编写Windows Store应用程序，你甚至都无法使用WebClient。

巧妙之处在于，方法在执行到await表达式时就返回了。在此之前，它与其他事件处理程序一样，都是在UI线程同步执行的。如果在第一行加断点进行调试，你会发现堆栈跟踪显示按钮正在触发Click事件，包括Button.OnClick方法。到达await后，代码将检查其结果是否存在。如果不存在（几乎总是如此），会安排一个在Web操作完成时将要执行的后续操作（continuation）。在本例中，后续操作将执行剩下的代码，跳到await表达式的末尾，并如你所愿地回到UI线程，以便在UI上进行操作。

说明　后续操作　后续操作指在异步操作（或任何Task）完成时执行的回调程序。在异步方法中，后续操作保留了方法的控制状态。就像闭包保留了环境中的变量一样，后续操作记住了它的位置，因此在执行时可回到原处。Task类包含一个专门用于添加后续操作的方法，即Task.ContinueWith。

如果在await表达式后的代码中加入断点（假设await表达式需要后续操作），你会发现堆栈跟踪中不再拥有Button.OnClick方法，该方法早已执行完毕。现在的调用栈是纯粹的Windows Forms事件循环，在顶部还有一些异步基本结构。如果为了适时地更新UI，而在后台线程中调用Control.Invoke，将得到跟现在非常类似的调用栈，然而现在这些工作已经完全为我们做好了。起初发现调用栈在眼皮底下发生如此显著的变化时，你可能会有些沮丧，但这对异步编程来说是绝对必要的。

你猜对了，所有这些都是因为编译器创建了一个复杂的状态机。这属于实现细节，并且对把握原理是非常有意义的。但首先我们需要一个更加具体的阐述：我们要得到什么，语言真正给了我们什么。

15.2　思考异步编程

如果让一个开发者描述异步执行过程，他很可能会谈起多线程。尽管这是异步编程的典型用途，但它却并不一定需要异步执行。要充分了解C# 5的异步特性是如何工作的，最好摒弃任何线程思想，然后回归基础。

15.2.1　异步执行的基础

异步编程与C#开发者所熟悉的执行模型不同。考虑以下简单代码：

```
Console.WriteLine("First");
Console.WriteLine("Second");
```

我们希望第一个调用完成后，再开始第二个调用。执行流从一条语句到下一条语句，按顺序执行。但异步执行模型不是这样。相反，它充斥了后续操作。在开始做一件事情的时候，要告知

其操作完成后应进行哪些操作。你应该听说过（或使用过）回调函数，二者理念相同，但它比我们这里讨论的范围要广泛。在异步编程上下文中，我使用该术语来表示保存程序控制状态的回调函数，而不是用于其他用途的回调函数，如GUI事件处理程序。

在.NET中，后续操作很自然地由委托加以表示，且通常为接收异步操作结果的action。因此，在C# 5之前使用`WebClient`中的异步方法，需要包装多个事件，来表示在成功或失败等情况下应该执行哪段代码。但问题是，即使可以使用Lambda表达式，为这一系列复杂的步骤创建委托，仍然是一件困难的事。如果想要验证错误处理是否正确，情况会变得更糟。（我对正确地编写异步代码的成功路径充满信心，但却不那么确定它在错误时能够正确地响应。）

实际上，C#编译器会对所有`await`都构建一个后续操作。这个理念表述起来非常简单，显然是为了可读性和开发者的健康。

我前面对异步编程的描述是理想化的。实际上基于任务的异步模式要稍有不同。它并不会将后续操作传递给异步操作，而是在异步操作开始时返回一个token，我们可以用这个token在稍后提供后续操作。它表示正在进行的操作，在返回调用代码前可能已经完成，也可能正在处理。token用于表达这样的想法：在这个操作完成之前，不能进行下一步处理。token的形式通常为`Task`或`Task<TResult>`，但这并不是必须的。

在C# 5中，异步方法的执行流通常遵守下列流程。

(1) 执行某些操作。

(2) 开始异步操作，并记住返回的token。

(3) 可能会执行其他操作。（在异步操作完成前，往往不能进行任何操作，此时忽略该步骤。）

(4) 等待异步操作完成（通过token）。

(5) 执行其他操作。

(6) 完成。

如果你不在乎“等待”，那么也可以在C# 4中完成这一切操作。如果你能接受在异步操作完成前进行阻塞，那么可以使用token。对于`Task`，可以调用`Wait()`。但这时，我们占用了一个有价值的资源（线程），却没有进行任何有用的工作。这就好像打电话订了比萨，然后就站在门口干等。你真正想要的，是做一些其他事情，在送货小哥到来前先忘掉比萨这码事儿。这时就需要`await`了。

在我们“等待”一个异步操作时，其实是在说，“现在我已经走得够远了，等异步操作完成后再继续前进。”但如果不想阻塞线程，又该怎么做呢？很简单，我们可以立即返回，然后异步地继续执行其他操作。如果想让调用者知道什么时候异步方法能够完成，就要传一个token回去，它们可以选择阻塞，或（更有可能）使用一个后续操作。通常，我们最终会得到一整栈互相调用的异步方法，就好像为了一段代码进入了“异步模式”（async mode）。C#从来没说过必须这么做，但消费异步操作的代码也表现得像异步操作，这无形中鼓励了此种做法。

同步上下文

之前我提到过，UI代码的金科玉律之一是，除非在正确的线程中，否则不要更新用户界面。在“检查页面长度”的示例中（代码清单15-1），我们需要确保await表达式之后的代码在UI线程上的执行。异步函数能够回到正确的线程中，是因为使用了SynchronizationContext类。该类早在.NET 2.0中就已存在，用以供BackgroundWorker等其他组件使用。SynchronizationContext涵盖了“在适当的线程上”执行委托这一理念。其Post（异步）和Send（同步）消息的方法，与Windows Forms中的Control.BeginInvoke和Control.Invoke异曲同工。

不同的执行环境使用不同的上下文。例如，某个上下文可能会从线程池中取出一个线程并执行给定的行为。除了同步上下文以外还有很多上下文信息，但如果你想知道异步方法是如何在正确的位置上执行的，就要牢记同步上下文。

要了解更多关于SynchronizationContext的信息，请阅读Stephen Cleary在MSDN杂志上关于该话题的文章（http://mng.bz/5cDw）。如果你是ASP.NET开发者的话，应尤其注意：ASP.NET上下文会让看上去没问题的代码死锁，轻易地让粗心的开发者掉进陷阱。

理论介绍完毕后，该深入地看看异步方法的具体细节了。异步匿名函数也适用于同样的模型，但异步方法介绍起来要简单多了。

15.2.2　异步方法

按图15-1所示来思考异步方法是非常有用的。

图中共有三个代码块（方法）和两个边界（方法返回类型）。以下为简单示例的具体代码：

```
static async Task<int> GetPageLengthAsync(string url)
{
   using (HttpClient client = new HttpClient())
   {
      Task<string> fetchTextTask = client.GetStringAsync(url);
      int length = (await fetchTextTask).Length;
      return length;
   }
}

static void PrintPageLength()
{
   Task<int> lengthTask =
      GetPageLengthAsync("http://csharpindepth.com");
   Console.WriteLine(lengthTask.Result);
}
```

图15-1的5个部分与上述代码的对应关系为：

- 调用方法为PrintPageLength；
- 异步方法为GetPageLengthAsync；
- 异步操作为HttpClient.GetStringAsync；

- 调用方法和异步方法之间的边界为Task<int>;
- 异步方法和异步操作之间的边界为Task<string>。

我们主要感兴趣的是异步方法本身,但也包含了其他方法,这样就能看到它们是如何交互的。特别是，你一定要了解方法边界处的有效类型。

我会在本章后续内容里经常提到这些块和边界，所以请把图15-1牢记在心。

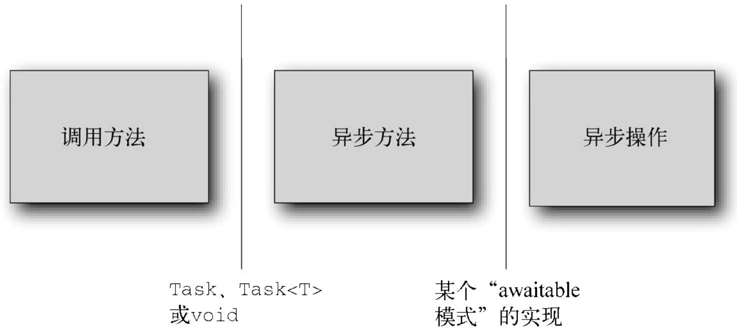


图15-1 异步模型

15.3 语法和语义

我们终于可以开始了解如何编写异步方法及其行为了。本节涵盖的范围很广，因为会在很大程度上将“能做什么”和“做时会发生什么”融合在一起。

新的语法只有两个：async是在声明异步方法时使用的修饰符，await表达式则负责消费异步操作。不过很快，信息在程序的不同部分间进行转移的方法就会成为难点，特别是在考虑到错误发生时的情况下更是如此。我试图将不同的部分分开来讲，但展示的代码会涉及所有内容。如果在阅读本节时想问“但是，那个是怎么回事？”的话，请继续读下去，问题可能马上就会得到解决。

我们从方法声明本身开始，这是最简单的部分。

15.3.1 声明异步方法

异步方法的声明语法与其他方法完全一样，只是要包含async上下文关键字。async可以出现在返回类型之前的任何位置。以下这些都是有效的：

```
public static async Task<int> FooAsync() { ... }
public async static Task<int> FooAsync() { ... }
async public Task<int> FooAsync() { ... }
public async virtual Task<int> FooAsync() { ... }
```

我个人喜欢将async修饰符放在返回类型前面，但你完全可以按自己的喜好来写。和以前一样，请跟团队成员讨论，并在一个代码库里保持一致。

async上下文关键字有一个不为人知的秘密：对语言设计者来说，方法签名中有没有该关键字都无所谓。就像在方法内使用具有适当返回类型的yield return或yield break，会使编译器进入某种“迭代器块模式”（iterator block mode）一样，编译器也会发现方法内包含await，并进入“异步模式”（async mode）。但我个人倾向于必须写async，因为它大大提高了异步方法代码的可读性。它明确表达了你的预期，你可以主动寻找await表达式，也可以寻找应该转换成异步调用和await表达式的块调用。

不过，async修饰符在生成的代码中没有作用[1]，这个事实是非常重要的。对调用方法来说，它只是一个可能会返回任务的普通方法。你可以将一个（具有适当签名的）已有方法改成使用async，反之亦然。对于源代码和二进制来说，这都是一个兼容的转换。

15.3.2　异步方法的返回类型

调用者和异步方法之间是通过返回值来通信的。异步函数的返回类型只能为：

- void；
- Task；
- Task<TResult>（某些类型的TResult，其自身即可为类型参数）。

.NET 4中的Task和Task<TResult>类型都表示一个可能还未完成的操作。Task<TResult>继承自Task。二者的区别是，Task<TResult>表示一个返回值为T类型的操作，而Task则不需要产生返回值。尽管如此，返回Task仍然很有用，因为调用者可以在返回的任务上，根据任务执行的情况（成功或失败），附加自己的后续操作。在某种意义上，你可以认为Task就是Task<void>类型，如果这么写合法的话。

之所以将异步方法设计为可以返回void，是为了和事件处理程序兼容。例如，可以像下面这样编写一个UI按钮点击处理程序：

```
private async void LoadStockPrice(object sender, EventArgs e)
{
    string ticker = tickerInput.Text;
    decimal price = await stockPriceService.FetchPriceAsync(ticker);
    priceDisplay.Text = price.ToString("c");
}
```

这是一个异步方法，但调用代码（按钮的OnClick方法，或其他触发该事件的框架代码）却并不真正关心这一点。它们只调用给定的事件处理程序，而没有必要知道事件什么时候真正处理完毕（例如加载完股票价格并更新UI）。编译器生成的代码将包含一个状态机，并且在FetchPriceAsync的返回值上附加一个后续操作，所有这一切都是实现细节。

可以让某个事件订阅上述方法，就像订阅其他事件处理程序一样：

```
loadStockPriceButton.Click += LoadStockPrice;
```

---

① 在某种意义上来说是这样。实际上后面我们会看到，它会给方法应用一个特性，但这并不是方法签名的一部分，并且对开发者来说是可以忽略的。它只是可以帮助工具找到“真正”的代码去哪儿了。

毕竟（是的，我一直在刻意强调这一点），对于调用代码来说，它只是一个普通方法，包含void返回类型以及object和EvengArgs类型的参数，这使得它能够作为一个EventHandler委托实例的行为。

对于一个异步方法，只有在作为事件订阅者时才应该返回void。在其他不需要特定返回值的情况下，最好将方法声明为返回Task。这样，调用者可以等待操作完成，以及探测失败情况等。

还有一个关于异步方法签名的约束：所有参数都不能使用out或ref修饰符。这么做是有道理的，因为这些修饰符是用于将通信信息返回给调用代码的；而且在控制返回给调用者时，某些异步方法可能还没有开始执行，因此引用参数可能还没有赋值。当然，更奇怪的是：将局部变量作为实参传递给ref形参，异步方法可以在调用方法已经结束的情况下设置该变量。这并没有多大意义，所以编译器干脆禁止这么做。

声明方法后，我们就可以编写方法体并等待其他异步操作了。

15.3.3　可等待模式

异步方法几乎包含所有常规C#方法所包含的内容，只是多了一个await表达式。我们可以使用任意控制流：循环、异常、using语句等。代码表现正常，唯一有趣的地方是await表达式的用途，以及返回值是如何传递的。

await的约束

与yield return一样，使用await表达式也有一些约束条件。它不能在catch或finally块、非异步匿名函数[①]、lock语句块或不安全代码中使用。

这些约束条件是为了保证安全，特别是关于锁的约束。如果你希望在异步操作完成时持有锁，那么应该重新设计你的代码。不要通过在try/finally块中手动调用Monitor.TryEnter和Monitor.Exit的方式绕过编译器的限制，而应该实现代码的更改，这样在操作过程中就不再需要锁了。如果此时的情况不允许代码的改变，则可考虑使用SemaphoreSlim和它的WaitAsync方法来代替。

await表达式非常简单，只是在其他表达式前面加了一个await。当然，对于能等待的东西是有限制的。需要提醒的是，我们正在谈论图15-1的第二个边界，即异步方法如何与其他异步操作交互。一般来说，我们只能等待（await）一个异步操作。换句话说，是包含以下含义的操作：

- 告知是否已经完成；
- 如未完成可附加后续操作；
- 获取结果，该结果可能为返回值，但至少可以指明成功或失败。

15

① 即没有用async声明的Lambda表达式和匿名方法，亦即所有在C# 4中有效的匿名函数声明。15.4节将讨论异步匿名函数。

你可能以为应该通过接口来表示，但（大多情况下）并非如此。这里只涉及一个接口，并且只涵盖了“附加后续操作”这一部分。这个接口十分简单，你甚至都不需要直接使用。它位于System.Runtime.CompilerServices命名空间，如下所示：

```
// 位于System.Runtime.CompilerServices的真正接口
public interface INotifyCompletion
{
   void OnCompleted(Action continuation);
}
```

大量工作都是通过模式来表示的，这有点类似于foreach和LINQ查询。为了更清晰地描述该模式的轮廓，假设存在一些相关的接口（但实际并没有）。稍后我会介绍真实情况，现在先来看看虚构的接口：

```
// 警告：这些并不存在
// 为包含返回值的异步操作建立的虚拟接口
public interface IAwaitable<T>
{
   IAwaiter<T> GetAwaiter();
}

public interface IAwaiter<T> : INotifyCompletion
{
   bool IsCompleted { get; }
   T GetResult();

   // 从INotifyCompletion继承
   // void OnCompleted(Action continuation);
}

// 为没有返回值的异步操作建立的虚拟接口
public interface IAwaitable
{
   IAwaiter GetAwaiter();
}

public interface IAwaiter : INotifyCompletion
{
   bool IsCompleted { get; }
   void GetResult();

   // 从INotifyCompletion继承
   // void OnCompleted(Action continuation);
}
```

这可能让你想起了IEnumerable<T>和IEnumerator<T>。为在foreach循环中迭代一个集合，编译器生成的代码首先调用GetEnumerator()，然后使用MoveNext()和Current。同样地，在异步方法中，对于一个await表达式，编译器生成的代码会先调用GetAwaiter()，然后适时地使用awaiter的成员来等待结果。

C#编译器要求awaiter必须实现INotifyCompletion。这主要是由于效率的原因。一些编译器的预发布版本根本就没有这个接口。

编译器仅通过签名来检查所有其他成员。重要的是，GetAwaiter()方法本身并不一定是一

个标准的实例方法。它可以是await表达式中对象的扩展方法。IsCompleted和GetResult必须是GetAwaiter()方法返回类型上的真正成员，但不一定具有公共属性，只要能由包含await表达式的代码访问即可。

前面讲述了什么样的表达式可以作为await关键字的目标，不过整个表达式本身也同样拥有一个有趣的类型：如果GetResult()返回void，那么整个await表达式就没有类型，而只是一个独立的语句。否则，其类型与GetResult()的返回类型相同。

例如，Task<TResult>.GetAwaiter()返回一个TaskAwaiter<TResult>，其GetResult()方法返回TResult。（希望你不会感到奇怪。）根据await表达式的类型规则，我们可以编写这样的代码：

```
using (var client = new HttpClient())
{
   Task<string> task = client.GetStringAsync(...);
   string result = await task;
}
```

相比之下，静态的Task.Yield()方法返回一个YieldAwaitable，其GetAwaiter()方法返回YieldAwaitable.YieldAwaiter，而YieldAwaitable.YieldAwaiter又包含一个返回void的GetResult方法。这意味着我们只能这样：

```
await Task.Yield();
```

或者，如果非要将二者分开的话（但是会很怪）：

```
YieldAwaitable yielder = Task.Yield();
await yielder;
```

这里的await表达式不会返回任何类型的值，因此不能将其分配给变量，或作为方法实参进行传递，也不能执行其他任何将表达式作为值的相关操作。

需要注意的是，由于Task和Task<TResult>都实现了可等待模式，因此可以在一个异步方法内调用另一个异步方法：

```
public async Task<int> FooAsync()
{
   string bar = await BarAsync();
   // 显然通常会更加复杂
   return bar.Length;
}
public async Task<string> BarAsync()
{
   // 一些异步代码，可能会调用更多的异步方法
}
```

组合异步操作正是异步特性大放异彩的一个方面。进入异步模式（mode）后，就可以很轻松地保持这种模式，编写自然流畅的代码。

但我抢先一步。我已经描述了await某些内容时编译器都需要什么，而不是编译器实际上会怎么做。

15.3.4　await表达式的流

await既是直观的，又是晦涩的，这是C# 5异步特性最奇特的一个方面。如果不太费神去思考，它就会显得非常简单。如果仅仅是接受目标的实现，而不去明确定义从哪里开始，那么一切就没有问题，至少在出错前没什么问题。

如果想弄明白究竟发生了什么才能达到预期效果，事情就变得有点棘手了。既然本书挂着“深入理解”的名头，那么我就假设你想知道这些细节。最终，我保证你在用await时会更加自信，并且更加高效。

即便如此，我还是建议各位根据自身情况，在两个不同级别发展阅读异步代码的能力。如果无须理解这里列出的单独步骤，就让它们随风而去好了。你可以像阅读同步代码那样去阅读异步代码，只需留意代码异步等待某些操作完成时的位置即可。然后，当遇到代码不按预期执行这种棘手问题时，可深入研究一下哪些地方涉及了哪些线程，以及调用栈在任意时间点的样子。（我并没有说这很简单，但理解其机制至少会对我们有所帮助。）

1. 展开复杂的表达式

首先来进行一些简化。await后面有时是方法调用的结果，有时是属性，如下所示[①]：

```
string pageText = await new HttpClient().GetStringAsync(url);
```

这看上去像是await可以修改整个表达式的含义一样。但其实await只是在操作一个值。上面的代码跟下面的是等价的：

```
Task<string> task = new HttpClient().GetStringAsync(url);
string pageText = await task;
```

同样地，await表达式的结果也可以用作方法实参，或作为其他表达式的一部分。也可以将await指定的部分从整体中分开，这样会有助于你的理解。

假设有两个方法GetHourlyRateAsync()和GetHoursWorkedAsync()，分别返回Task<decimal>和Task<int>。那么很可能会产生以下复杂语句：

```
AddPayment(await employee.GetHourlyRateAsync() *
           await timeSheet.GetHoursWorkedAsync(employee.Id));
```

这里会应用基本的C#表达式求值规则，*左边操作数的求值发生在右边操作数的求值之前，因此上面的语句可以展开为如下形式：

```
Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
decimal hourlyRate = await hourlyRateTask;
Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
int hoursWorked = await hoursWorkedTask;
AddPayment(hourlyRate * hoursWorked);
```

这种展开形式暴露了原始语句的效率问题。可在等待任务启动前，通过调用GetHourlyRateAsync()和GetHoursWorkedAsync()方法启动这两个任务，以便在代码中引入并行操作。

就目前来看，更有用的结论是，你只需要在某个值的上下文中检查await的行为即可。即使

① 这个示例不太恰当，因为我们通常会对HttpClient使用using语句。希望你们原谅我没有释放资源，只此一次。

该值源自一个方法调用，但由于我们谈论的是异步，所以可以忽略这个方法调用。

2. 可见的行为

执行过程到达await表达式后，存在着两种可能：等待中的异步操作已经完成，或还未完成。

如果操作已经完成，那么执行流程就非常简单，只需继续执行即可。如果操作失败，并且由一个代表该失败的异常所捕获，则会抛出该异常。否则，将得到该操作所返回的结果，例如从`Task<string>`中提取的`string`，然后继续执行程序的下一部分。所有这一切，都无需任何线程上下文切换或附加任何后续操作。

你也许会问，为什么可以立即完成的操作会首先用异步来表示。这有点类似于在LINQ序列上调用`Count()`方法：一般情况下可能需要迭代序列中的每个元素，但在某些情况下（如序列为`List<T>`）可以做简单的优化。如果能有一种抽象来覆盖这两种场景，而不必考虑执行时间时，这将是非常有用的。举个真实的例子，在异步API中，从与磁盘文件相关联的流中异步读取数据。也许作为之前`ReadAsync`调用请求的结果，想要读取的所有数据已经从磁盘获取到了内存中，那么显然应该立即使用这些数据，而不再继续剩下的异步机制。

更有趣的场景发生在异步操作仍在执行时。在这种情况下，方法异步地等待操作完成，然后继续执行适当的上下文。这种“异步等待”意味着方法将不再执行，它把后续操作附加在了异步操作上，然后返回。异步操作确保该方法在正确的线程中恢复。其中正确的线程通常指线程池线程（具体使用哪个线程都无妨）或UI线程。

从开发者的角度来看，感觉像是方法在异步操作完成时就暂停了。就方法中使用的所有局部变量而言，编译器应确保其变量值在后续操作开始前后均保持不变，就像在迭代器块中一样。

图15-2描述了这一流程，尽管传统的流程图并不是为异步行为而设计的。

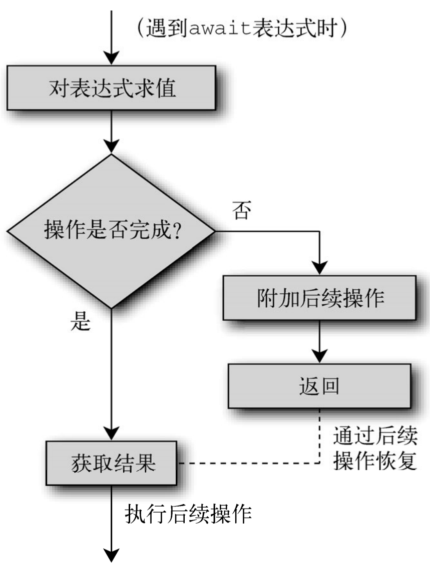


图15-2 await处理的用户可见模型

你可以把虚线看作是来自流程图顶部的另一条线。注意，这里我假设await表达式的目标包含一个结果。如果等待的只是一个普通的Task或类似的对象，则“获取结果”实际上意味着“检查操作是否成功完成”。

现在，有必要停下来略微思考一下，从一个异步方法“返回”意味着什么。同样，这里也存在着两种可能。

- 这是你需要等待的第一个await表达式，因此原始调用者还位于栈中的某个位置。（记住，在到达需要等待的操作之前，方法都是同步执行的。）
- 已经等待了其他操作，因此处于由某个操作调用的后续操作中。调用栈与第一次进入该方法时相比，已经发生了翻天覆地的变化。

在第一种情况下，最终往往会将Task或Task<T>返回给调用者。显然，这时还不能得到方法的真实结果，因为即使没有返回值，也无法得知方法的完成是否存在异常。因此，需要返回的任务必须是未完成的。

在后一种情况下，“某些操作”的回调取决于你的上下文。例如，在Windows Forms UI中，如果在UI线程上启动异步方法，并且不故意进行切换，那么整个方法都将在UI线程上执行。在方法的第一部分，你将位于某个事件处理程序，或其他启动异步方法的地方。然后，可直接由消息泵进行回调，就像使用Control.BeginInvoke(continuation)一样。这时，调用代码（不管是Windows Forms消息泵、线程池机制的一部分，还是别的什么）并不关心你的任务。

注意，在遇到第一个真正的异步await表达式之前，方法的执行是完全同步的。调用异步方法，与在单独的线程中启动一个新任务不同，并且你应确保总是编写能够快速返回的异步方法。当然，这取决于所写代码的上下文，但一般应避免在异步方法中执行耗时的工作。而应将其分离到其他方法，并为其创建一个Task。

3. 使用可等待模式的成员

了解过要实现的内容后，如何使用可等待模式的成员就相当简单了。图15-3与图15-2非常相似，唯一不同的是本图包含了调用该模式的代码。

若写成这样，你可能会问为何要如此兴师动众，有必要提供语言支持吗？不过，附加一个后续操作要比想象中复杂得多。在控制流都是线性的这种非常简单的情况下（处理一些操作，等待一些操作，再处理一些，再等待一些），可将后续操作想象成Lambda表达式，尽管并不是十分贴切。而只要代码中包含循环或条件判断，并且希望将代码包含在同一方法中，情况就会变得尤为复杂。这时C# 5的优势就体现了出来。尽管你可能会说，这不过是编译器的语法糖，但手工创建后续操作和让编译器创建相比，其可读性可谓天壤之别。

与自动实现属性等简单的转换不同，即使异步方法本身十分短小，编译器自动生成的代码也与手写的完全不同。在稍后的部分中我们会简要介绍这种转换，而你已经可以开始接触“幕后主使”了——希望异步方法现在对你而言已经不那么神秘了。

我们已经介绍了异步方法返回类型的限制，以及await表达式如何通过GetResult()方法解包异步操作的结果，但还没有提及二者间的关系，或者说如何从异步方法返回值。

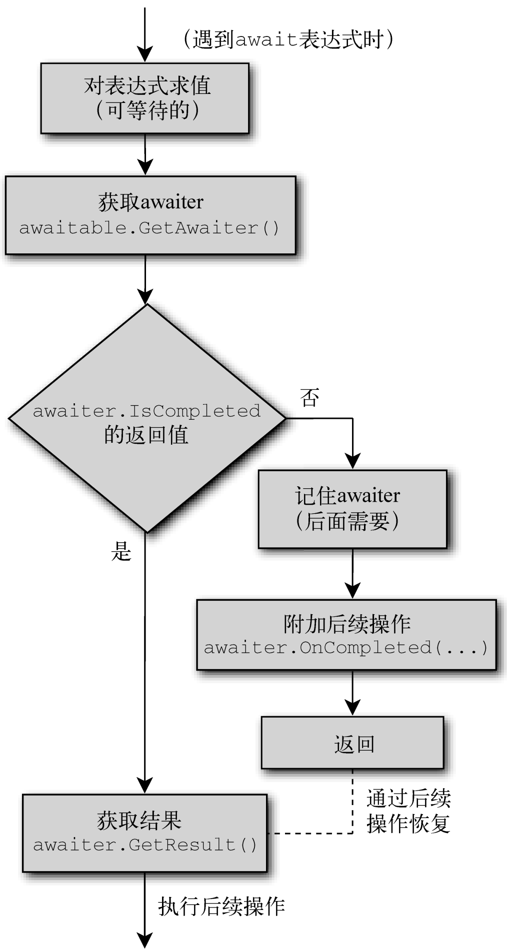


图15-3　通过可等待模式处理await

15.3.5　从异步方法返回

再来看一看这个返回数据的示例，这次我们只关注返回部分：

```
static async Task<int> GetPageLengthAsync(string url)
{
   using (HttpClient client = new HttpClient())
   {
      Task<string> fetchTextTask = client.GetStringAsync(url);
      int length = (await fetchTextTask).Length;
      return length;
   }
}
```

可以看到，length的类型为int，但方法的返回类型为Task<int>。生成的代码帮我们进

15

行了包装，因此调用者将得到一个Task<int>，并最终在方法完成时得到其返回值。只返回Task的方法，有点类似于普通的void方法，它不需要任何返回语句，如果有则必须为简单的return，而不能指定某个值。在这两种情况下，任务都会在异步方法中传播（propagate）抛出的异常。

希望现在你已经对这种包装的必要性有了准确的认识：在到达return语句之前，几乎必然会返回调用者，我们需以某种方式向这个调用者传播信息。一个Task<T>（即计算机科学中的future），是对未来生成的值或抛出的异常所做出的承诺（promise）。

和普通的执行流一样，如果return语句出现在有finally块的try块中（包括using语句），那么用来计算返回值的表达式将立即被求值，但直到所有对象清理完毕后，才会作为任务结果。这意味着如果finally块抛出一个异常，则整个代码都会失败。

再次强调一点之前提及的内容，即是自动包装（wrap）与拆包（unwrap）相结合，才使得异步特性工作得如此和谐。它与LINQ有些类似：在LINQ中，我们对序列的每个元素执行操作，而包装和拆包指的是，可将这些操作应用到序列，并返回这个序列。在异步世界里，你很少需要显式处理某个任务，而是await一个任务来进行消费，并作为异步方法机制的一部分，自动生成一个结果任务。

15.3.6　异常

当然，程序并不会总是执行得一帆风顺，.NET表示失败的惯用方式是使用异常。与向调用者返回值类似，异常处理需要语言的额外支持。在想要抛出异常时，异步方法的原始调用者可能已经不在栈上了；而当await的异步操作失败时，其原始调用者可能没有执行在同一条线程上，因此需要某种方式来封送（marshaling）失败。如将失败认作另一种形式的结果，可有助于理解异常和返回值采用类似处理机制的原因所在。

本节将介绍异常如何跨越图15-1中的两个边界。我们先从异步方法及其所等待的异步操作间的边界讲起。

1. 在等待时拆包异常

awaiter的GetResult方法可获取返回值（如果存在的话）；同样地，如果存在异常，它还负责将异常从异步操作传递回方法中。听上去简单做起来难，因为在异步世界里，单个Task可表示多个操作，并导致多个失败。尽管还存在其他的可等待模式实现，但有必要专门介绍Task，因为在大多数情况下，我们等待的都是这个类型。

Task有多种方式可以表示异常。

- 当异步操作失败时，任务的Status变为Faulted（并且IsFaulted返回true）。
- Exception属性返回一个AggregateException，该AggregateException包含所有（可能有多个）造成任务失败的异常；如果任务没有错误，则返回null。
- 如果任务的最终状态为错误，则Wait()方法将抛出一个AggregateException。
- Task<T>的Result属性（同样等待完成）也将抛出AggregateException。

此外，任务还支持取消操作，可通过CancellationTokenSource和CancellationToken

来实现这一点。如果任务取消了，Wait()方法和Result属性都将抛出包含OperationCanceledException的AggregateException（实际上是一个TaskCanceledException，它继承自OperationCanceledException），但状态将变为Canceled，而不是Faulted。

在等待任务时，任务出错或取消都将抛出异常，但并不是AggregateException。大多情况下为方便起见，抛出的是AggregateException中的第一个异常，往往这就是我们想要的。异步特性就是像编写同步代码那样编写异步代码，如下所示：

```
async Task<string> FetchFirstSuccessfulAsync(IEnumerable<string> urls)
{
    //TODO：验证是否获取到了URL
    foreach (string url in urls)
    {
        try
        {
            using (var client = new HttpClient())
            {
                return await client.GetStringAsync(url);
            }
        }
        catch (WebException exception)
        {
            // TODO：记录日志、更新统计信息等
        }
    }
    throw new WebException("No URLs succeeded");
}
```

目前，先不要在意损失所有的原始异常，以及按顺序获取所有页面。我想说明的是，我们希望在这里捕获WebException。执行一个使用HttpClient的异步操作，失败后可抛出WebException。我们想捕获并处理它，对吧？但GetStringAsync()方法不能为服务器超时等错误抛出WebException，因为方法仅仅启动了操作。它只能返回一个包含WebException的任务。如果简单地调用该任务的Wait()方法，将会抛出一个包含WebException的AggregateException。任务awaiter的GetResult方法将抛出WebException，并被以上代码所捕获。

当然，这样会丢失信息。如果错误的任务中包含多个异常，则GetResult只能抛出其中的一个异常（即第一个）。你可能需要重写以上代码，这样在发生错误时，调用者就可捕获AggregateException并检查所有失败的原因。重要的是，一些框架方法（如Task.WhenAll()）也可以实现这一点。WhenAll()方法可异步等待（方法调用中指定的）多个任务的完成。如果其中有失败的，则结果即为失败，并包含所有错误任务中的异常。但如果只是等待（await）WhenAll()返回的任务，则只能看到第一个异常。

幸好，要解决这个问题并不需要太多的工作。我们可以使用可等待模式的知识，编写一个Task<T>的扩展方法，从而创建一个可从任务中抛出原始AggregateException的特殊可等待模式成员。由于篇幅原因，此处不便给出完整代码，因此只能在代码清单15-12中列出其主旨内容。

代码清单15-2　重新包装任务失败时产生的多个异常

```
public static AggregatedExceptionAwaitable WithAggregatedExceptions(
                                                        this Task task)
{
   return new AggregatedExceptionAwaitable(task);
}

// In AggregatedExceptionAwaitable
public AggregatedExceptionAwaiter GetAwaiter()
{
   return new AggregatedExceptionAwaiter(task);
}

// In AggregatedExceptionAwaiter
public bool IsCompleted
{
     get { return task.GetAwaiter().IsCompleted; }      <─┐
}                                                          │
public void OnCompleted(Action continuation)               ❶ 委托给任务awaiter
{                                                          │
   task.GetAwaiter().OnCompleted(continuation);        <─┘
}
public void GetResult()
{                                          ❷ 发生错误时，直接抛出AggregateException
   task.Wait();                          <─┘
}
```

`Task<T>`也需要一个类似的方法，即在`GetResult()`中使用`return task.Result`，而不是调用`Wait()`。重点在于，我们把自己不想处理的部分委托给了任务的awaiter❶，而回避了`GetResult()`的常规行为，即对异常进行拆包。在调用`GetResult`时，我们知道该任务处于即将结束的状态，因此`Wait()`调用❷可立即返回，这并不妨碍我们要实现的异步性。

要使用这段代码，只需像代码清单15-2那样调用该扩展方法并等待结果即可。

代码清单15-3　捕获包含多个异常的AggregateException

```
private async static Task CatchMultipleExceptions()
{
   Task task1 = Task.Run(() => { throw new Exception("Message 1");
   });
   Task task2 = Task.Run(() => { throw new Exception("Message 2");
   });
   try
   {
      await Task.WhenAll(task1, task2).WithAggregatedExceptions();
   }
   catch (AggregateException e)
   {
      Console.WriteLine("Caught {0} exceptions: {1}",
         e.InnerExceptions.Count,
         string.Join(", ",
            e.InnerExceptions.Select(x => x.Message)));
   }
}
```

WithAggregateException()返回自定义的可等待模式成员，而后者的GetAwaiter()又提供自定义的awaiter，并支持C#编译器所需要的操作来等待结果。注意，也可将可等待模式成员和awaiter合并，并没有要求二者必须是不同类型，但分开的话会感觉更清晰一些。

代码清单15-3的输出如下所示：

```
Caught 2 exceptions: Message 1, Message 2
```

相对而言，我们很少需要这么做，少到微软没有在框架中提供任何支持，但知道这种方式还是很有必要的。

有关第二个边界的异常处理，我们知道这么多就足够了，至少现在来说是这样。但位于异步方法和调用者之间的第一个边界呢？

2. 在抛出异常时进行包装

你可能已经猜到我要说什么了，没错，就是异步方法在调用时永远不会直接抛出异常。异常方法会返回Task或Task<T>，方法内抛出的任何异常（包括从其他同步或异步操作中传播过来的异常）都将简单地传递给任务，就像前面介绍的那样。如果调用者直接等待[①]任务，则可得到一个包含真正异常的AggregateException；但如果调用者使用await，异常则会从任务中解包。返回void的异步方法可向原始的SynchronizationContext报告异常，如何处理将取决于上下文[②]。

除非你真的在乎为特定的上下文包装和解包异常，否则只需捕获嵌套的异步方法所抛出的异常即可。代码清单15-4证实了这么做是多么眼熟。

代码清单15-4　以熟悉的方式处理异步的异常

```
static async Task MainAsync()
{
   Task<string> task = ReadFileAsync("garbage file");      ❶ 开始异步读取
   try
   {
      string text = await task;                            ❷ 等待内容
         Console.WriteLine("File contents: {0}", text);
   }
   catch (IOException e)                    ❸ 处理IO失败
   {
      Console.WriteLine("Caught IOException: {0}", e.Message);
   }
}

static async Task<string> ReadFileAsync(string filename)
{
   using (var reader = File.OpenText(filename))            ❹ 同步打开文件
   {
      return await reader.ReadToEndAsync();
   }
}
```

① 这里的“等待”为wait，即调用Wait()方法，与前面提到的“等待”不同，后者指的是使用await关键字。若不加特殊说明，本章的“等待”均为使用await关键字。——译者注

② 15.6.4节将介绍有关上下文的更多细节。

调用File.OpenText时可抛出一个IOException❹（除非创建了一个名为“garbage file”的文件），但如果ReadToEndAsync返回的任务失败了，也会出现同样的执行路径。在MainAsync中，ReadFileAsync的调用❶发生在进入try块之前，但只有在等待任务时❷，调用者才能看到异常并在catch块中捕获❸，就像前面的WebException示例一样。同样，除异常发生的时机以外，其行为我们也非常熟悉。

与迭代器块类似，参数验证会有些麻烦。假设我们在验证完参数不含有空值后，想在异步方法里做一些处理。如果像在同步代码中那样验证参数，那么在等待任务之前，调用者不会得到任何错误提示。代码清单15-5给出了一个这样的例子。

代码清单15-5　异步方法中失效的参数验证

```
static async Task MainAsync()
{
   Task<int> task = ComputeLengthAsync(null);        ← 故意传入错误的参数
   Console.WriteLine("Fetched the task");
   int length = await task;                          ← ❶ 等待结果
   Console.WriteLine("Length: {0}", length);
}
static async Task<int> ComputeLengthAsync(string text)
{
   if (text == null)
   {
      throw new ArgumentNullException("text");       ← ❷ 立即抛出异常
   }
   await Task.Delay(500);                            ← 模拟真实的异步工作
   return text.Length;
}
```

代码清单15-5在失败前会先输出Fetched the task。实际上，在输出这条结果之前，异常就已经同步地抛出了，这是因为在验证语句之前并不存在await表达式❷。但调用代码直到等待返回的任务时❶，才能看到这个异常。一般来说，参数验证无须耗时太久（或导致其他异步操作）。最好能在系统陷入更大的麻烦以前，立即报告失败的存在。例如，如果对HttpClient.GetStringAsync传递一个空引用，则其可立即抛出异常。

在C# 5中，有两种方式可以迫使异常立即抛出。第一种方式是将参数验证和实现分离，这与代码清单6-9中处理迭代器块的情形相同。以下代码清单展示了ComputeLengthAsync的固定版本。

代码清单15-6　将参数验证从异步实现中分离出来

```
static Task<int> ComputeLengthAsync(string text)
{
   if (text == null)
   {
      throw new ArgumentNullException("text");
   }
   return ComputeLengthAsyncImpl(text);
```

```
}

static async Task<int> ComputeLengthAsyncImpl(string text)
{
    await Task.Delay(500); // 模拟真正的异步工作
    return text.Length;
}
```

在代码清单15-6中，就语言形式而言，ComputeLengthAsync本身并不是一个异步方法，因为它没有async修饰符。该方法执行时使用的是正常的执行流，因此如果方法开始处的参数验证抛出异常，就真的会抛出异常。而如果通过验证，则返回ComputeLengthAsyncImpl方法（工作真正发生的地方）创建的任务。在更现实的场景中，ComputeLengthAsync可以为公共或内部（internal）方法，而ComputeLengthAsyncImpl应该为私有方法，因为它假设参数验证已经执行过了。

另一个及早（eager）验证的方法是使用异步匿名函数，15.4节再来讨论这个示例。

异步方法中还有一种异常，其处理方式与其他异常不同，这个异常就是：取消（cancellation）。

3. 处理取消

任务并行库（TPL）利用CancellationTokenSource和CancellationToken两种类型向.NET 4中引入了一套统一的取消模型。该模型的理念是，创建一个CancellationTokenSource，然后向其请求一个CancellationToken，并传递给异步操作。可在source上只执行取消操作，但该操作会反映到token上。（这意味着你可以向多个操作传递相同的token，而不用担心它们之间会相互干扰。）取消token有很多种方式，最常用的是调用ThrowIfCancellationRequested，如果取消了token，并且没有其他操作，则会抛出OperationCanceledException。如果在同步调用（如Task.Wait）中执行了取消操作，则可抛出同样的异常。

C# 5规范中并没有说明取消操作如何与异步方法交互。根据规范，如果异步方法体抛出任何异常，该方法返回的任务则将处于错误状态。“错误”的确切含义因实现而异，但实际上，如果异步方法抛出OperationCanceledException（或其派生类，如TaskCanceledException），则返回的任务最终状态为Canceled。以下代码清单证实了导致任务取消的原因确实是一个异常。

代码清单15-7　通过抛出OperationCanceledException来创建一个取消的任务

```
static async Task ThrowCancellationException()
{
    throw new OperationCanceledException();
}
...
Task task = ThrowCancellationException();
Console.WriteLine(task.Status);
```

这段代码的输出为Canceld，而不是Faulted。如果在任务上执行Wait()，或请求其结果（针对Task<T>），则AggregateException内还是会抛出异常，所以没有必要在每次使用任务时都显式检查是否有取消操作。

说明　竞态会挂吗？　你可能会问，如果代码清单15-7中存在竞态条件会怎样。毕竟，我们调用的是异步方法，并立即获取任务的状态。如果真的开启了一个新的线程，是十分危险的。但其实并非如此。记住，在第一次到达await表达式之前，异步方法都是同步执行的。它仍然会包装结果和异常，但异步方法并不一定意味着存在多个线程。在示例中，ThrowCancellationException方法内没有任何await表达式，因此整个方法（虽然只有一行）都是同步运行的；你只是知道在它返回时可得到一个结果而已。Visual Studio会警告你"异步方法没有await表达式"，但在本例中这正是我们想要的。

重要的是，等待一个取消了的操作，将抛出原始的OperationCanceledException。这意味着如果不采取一些直接的行动，从异步方法返回的任务同样会被取消，因为取消操作具有可传播性。

代码清单15-8给出了一个有关任务取消操作的更为实际的例子。

代码清单15-8　通过一个取消的延迟操作来取消异步方法

```
static async Task DelayFor30Seconds(CancellationToken token)
{
   Console.WriteLine("Waiting for 30 seconds...");
   await Task.Delay(TimeSpan.FromSeconds(30), token);        ❶ 启动一个异步的延迟操作
}
...
var source = new CancellationTokenSource();
var task = DelayFor30Seconds(source.Token);                  ❷ 调用异步方法
source.CancelAfter(TimeSpan.FromSeconds(1));                 ❸ 请求延迟的token取消操作
Console.WriteLine("Initial status: {0}", task.Status);
try
{
   task.Wait();                                              ❹ 等待完成（同步）
}
catch (AggregateException e)
{
   Console.WriteLine("Caught {0}", e.InnerExceptions[0]);
}
Console.WriteLine("Final status: {0}", task.Status);         ❺ 显示任务状态
```

上述代码中启动了一个异步操作❷，该操作调用Task.Delay模拟真正的工作❶，并提供了一个CancellationToken。这一次，我们的确涉及了多个线程：到达await表达式时，控制返回到调用方法，这时要求CancellationToken在1秒后取消❸。然后（同步地）等待任务完成❹，并期望在最终得到一个异常。最后展示任务的状态❺。

代码清单15-8的输出结果如下所示：

```
Waiting for 30 seconds...
Initial status: WaitingForActivation
Caught System.Threading.Tasks.TaskCanceledException: A task was canceled.
Final status: Canceled
```

可认为取消操作默认是可传递的：如果A操作等待B操作，而B操作被取消了，那么我们认为A操作也被取消了。

当然，你不必这么做。你可以在DelayFor30Seconds方法中捕获OperationCanceledException，然后或继续做其他事情，或立即返回，或干脆抛出一个其他类型的异常。异步特性不会移除控制，它只是提供了一种有用的默认行为而已。

说明　**小心使用该代码**　代码清单15-8在控制台程序中或从线程池线程调用时，均可运作良好。但如果在Windows Forms UI线程（或其他单线程同步上下文）上执行这段代码，则会造成死锁。能看出原因吗？想想在延迟任务完成时，DelayFor30Seconds方法会试图返回到哪个线程上？再想想task.Wait()调用运行在哪个线程上？这是个相对简单的例子，但一些程序员在初次接触异步代码时往往会犯同样的错误。从根本上来说，问题在于调用了Wait()方法或Result属性。在相关任务完成前，二者均可阻塞线程。我并不是说不能使用它们，但在每次使用时必须考虑清楚。我们应该总是使用await，来异步地等待任务的结果。

以上内容很好地涵盖了异步方法的行为。我们在C# 5中使用的异步特性，大多是通过异步方法实现的。不过异步方法还有一个近亲……

15.4　异步匿名函数

我们不会在异步匿名函数上着太多笔墨。和你想象的一样，它是两个特性的结合体，即匿名函数（Lambda表达式和匿名方法）和异步函数（包含await表达式的代码）。基本上可以通过异步匿名函数来创建表示异步操作的委托[①]。目前我们所学的所有关于异步方法的知识，均适用于异步匿名函数。

创建异步匿名函数，与创建其他匿名方法或Lambda表达式类似，不同的是要在前面加上async修饰符。例如：

```
Func<Task> lambda = async () => await Task.Delay(1000);
Func<Task<int>> anonMethod = async delegate()
{
   Console.WriteLine("Started");
   await Task.Delay(1000);
   Console.WriteLine("Finished");
   return 10;
};
```

与异步方法一样，在创建委托时，委托签名的返回类型必须为void、Task或Task<T>。与其他匿名函数一样，你可以捕获变量，也可以添加参数。同样，异步操作的启动发生在委托的调

15

① 不能使用异步匿名函数创建表达式树。

用之后，并且多次调用会创建多个操作。委托调用会开启一个异步操作。与异步方法一样，开启异步操作的并不是await，也不是非要对异步匿名函数的结果使用await。

代码清单15-9展示了一个简单但完整的示例（尽管还是没有实际意义）。

代码清单15-9　使用Lambda表达式创建并调用一个异步函数

```
Func<int, Task<int>> function = async x =>
{
   Console.WriteLine("Starting... x={0}", x);
   await Task.Delay(x * 1000);
   Console.WriteLine("Finished... x={0}", x);
   return x * 2;
};

Task<int> first = function(5);
Task<int> second = function(3);
Console.WriteLine("First result: {0}", first.Result);
Console.WriteLine("Second result: {0}", second.Result);
```

此处我故意选择这样的值，以便让第二个操作早于第一个完成。但由于我们要在等待第一个操作完成后再打印结果（使用Result属性，这将阻塞线程直到任务结束。再次强调一遍，运行这样的代码时要十分谨慎！），因此输出结果如下：

```
Starting... x=5
Starting... x=3
Finished... x=3
Finished... x=5
First result: 10
Second result: 6
```

将异步代码放到异步方法中，也可得到同样的结果。

异步匿名函数并不会让我感到特别兴奋，但它也有自己的用途。尽管不能应用于LINQ查询表达式，但在某些情况下，还是可以实现数据转换的异步执行的。这时，只需以一种略微不同的方式思考整个过程即可。

在讨论完成分后，我们还会回到这个话题，但首先我想向大家展示一下异步匿名函数特别有用的一个方面。我之前承诺过，要展示另一种在异步方法开始时及早执行参数验证的方式。你可能还记得，在进入主操作前，需检查参数值是否为空。代码清单15-10是一个单个方法，其结果与代码清单15-6中两个分离方法得到的结果完全相同。

代码清单15-10　使用异步匿名函数进行参数验证

```
static Task<int> ComputeLengthAsync(string text)
{
   if (text == null)                                    ❶ 完全异步地验证
   {
      throw new ArgumentNullException("text");
   }
   Func<Task<int>> func = async () =>                   ❷ 创建一个异步函数
```

```
    {
        await Task.Delay(500);                    ◁── 模拟真正的异步工作
        return text.Length;
    };
    return func();                            ◁─❸ 调用匿名函数
}
```

你会发现这并不是一个异步方法。如果是的话，异常会被包装到任务里，而不是立即抛出。但我们还是想返回一个任务，因此在验证❶之后，将工作包装到一个异步匿名函数中❷，调用委托❸并返回结果。

尽管这看上去还是有点丑，但比分割成两个方法要清晰多了。不过性能上会蒙受一点损失：额外的包装会产生额外的代价。这在大多数情况下都没有问题，但如果你编写的库会应用于注重性能的程序，则应在真实场景中检测成本，然后再决定使用哪种方法。

说明 **VB的优势？** Visual Basic 11终于支持了C# 2就有的迭代器块。这种姗姗来迟反倒使团队有时间反思C#中的不足——Visual Basic实现方法支持匿名迭代器函数，并可在同种方法内将及早执行和延迟执行分离开来。而这个特性是C#所不具备的。

我们已经学习了C# 5异步特性的很多内容。接下来我们将深入部分实现细节，然后看看如何充分利用这个特性。所有这一切都以对前文内容的掌握为前提。如果你还没有尝试过示例代码（或最好是你自己的实验代码），那么现在是个绝佳时机。即使确认自己已经理解了理论，还是有必要写一些`async`和`await`，来真正感受一下什么是以同步的风格编写异步程序。

15.5 实现细节：编译器转换

我还清楚地记得2010年10月28日的那个晚上。Anders Hejlsberg在PDC演示`async/await`，而就在他演讲开始不久前，网上出现了大量的下载资源，包括C#规范变化部分的草案、CTP版本的C# 5编译器，以及Anders所演示的幻灯片。我在观看演讲现场的同时，一边浏览着幻灯片一边安装着CTP版本。在Anders结束演讲时，我已经在尝试编写异步代码了。

在接下来的几周内，我开始庖丁解牛——查看编译器生成的代码、为CTP版本的库编写简单的实现方法，并从各个角度深入探究。在新版本诞生时，我已经明确了具体的变动，并且对后台操作感到越发地习惯。了解得越多，就越感激编译器为我们生成的大量模板代码。这就像是在显微镜下观看美丽的花朵：美依然存在，但你却能从中发现当初未能发现的韵味。

当然，并非所有人都跟我一样。如果你只是想依赖我前面描述的行为，并单纯地相信编译器会执行正确的操作，那绝对没有问题。另外，如果暂时跳过该节，等以后再来学习这一部分，也不会错过什么内容。本章后续部分都与本节内容无关。你在调试代码时，不太可能会达到此处描述的级别。但我相信本节可使你更好地理解整个特性是如何融汇在一起的。查看生成代码后，可等待模式必然会变得更加容易理解，并且还会看到框架提供的一些用于帮助编译器的类型。一些

最恐怖的细节只有在优化时才会出现。例如，为避免不必要的堆分配和上下文切换，设计和实现进行了十分小心地调整。

作为粗略的描述，假设C#编译器将“使用async/await的C#代码”转换为“不使用async/await的C#代码”。实际上，我们无从知晓编译器的内部实现，并且这种转换很可能发生在比C#还要低的级别[①]内。生成的IL肯定无法全部使用非异步的C#来表述，因为C#对流控制的限制比IL要严格。但将其想象成是C#的功能要简单些，这有助于我们理解这些拼图是如何拼接在一起的。

生成代码和洋葱有点儿类似，其复杂程度由外到内，层层递增。首先从最外层讲起，然后逐渐深入，直至棘手的await表达式以及awaiter和后续操作的组合。

15.5.1　生成的代码

还在吗？我们开始吧。由于深入讲解需上百页的篇幅，因此这里我不会讲得太深。但我会提供足够的背景知识，以有助于你对整个结构的理解。之后可通过阅读我近些年来撰写的博客文章，来了解更加错综复杂的细节，或简单地编写一些异步代码并反编译。同样地，这里我只介绍异步方法，它包含了所有有趣的机制，并且不需要处理异步匿名函数所处的间接层。

说明　警告，勇敢的旅行者——前方是实现细节！　本节将描述微软C# 5编译器（随着.NET 4.5的发布而推出）内实现的相关内容。从CTP版到beta版，有些细节变化很大，并且在未来仍有可能发生改变。但我认为其基本理念并不会发生太大的变动。充分了解本节内容后，你会发现并不存在什么魔法，只不过是一些编译器生成的聪明代码罢了。这之后便可以从容应对未来变化的细节内容了。

正如我之前多次提到过的，它的实现（包括近似实现和真实编译器生成的代码）基本上可以说是一个状态机。编译器将生成一个私有的内嵌结构，来表示这个异步方法。这个结构还必须包含一个方法，其签名与所声明的方法签名相同。我称其为骨架方法，该方法本身没有多少内容，但其他东西都依赖于它。

骨架方法需要创建状态机，并执行一个步骤（此处的步骤指执行第一个await表达式之前的代码），然后返回一个表示状态机进度的任务。（别忘了，在第一次到达真正需要等待的await表达式之前，执行过程是同步的。）此后，骨架方法的运作就此结束。状态机会负责其余事项，后续操作附加到其他异步操作后，可通知状态机去执行另一个步骤。当之前返回的任务被赋予适当的值后，方法就执行到最后了，状态机可随即发出信号。图15-4展示了这一流程图。

---

① 如IL。——译者注

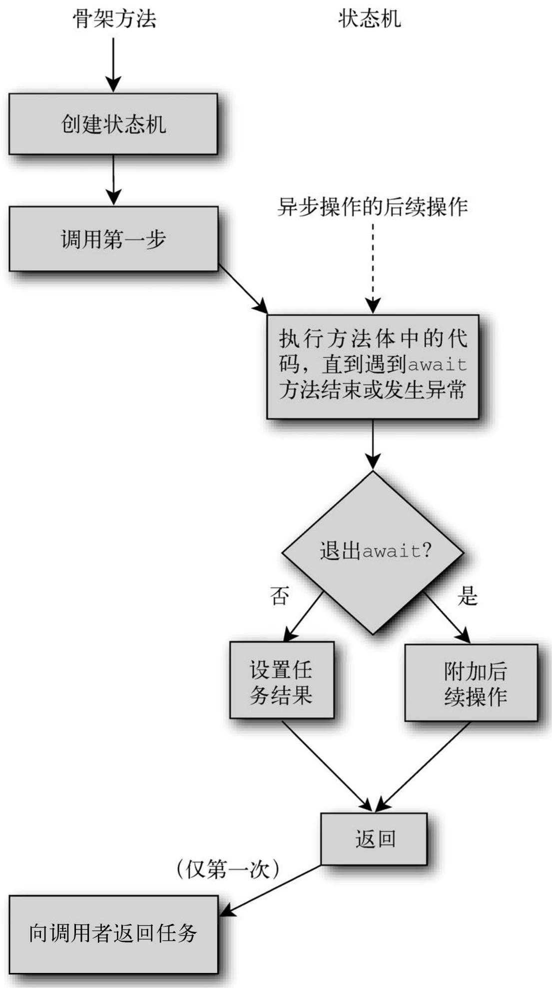


图15-4　生成代码的流程图

当然，"执行方法体中的代码"这一步，只有在骨架方法中第一次调用时，才会从方法的开头执行。以后每次到达该块，都是由后续操作从之前中断的地方开始继续执行。

现在有两个概念需要关注，即骨架方法和状态机。在本节的剩余篇幅中，我将使用单个异步方法作为示例，如代码清单15-11所示。

代码清单15-11　通过简单的异步方法来演示编译器转换

```
static async Task<int> SumCharactersAsync(IEnumerable<char> text)
{
   int total = 0;
   foreach (char ch in text)
   {
      int unicode = ch;
      await Task.Delay(unicode);
```

```
        total += unicode;
    }
    await Task.Yield();
    return total;
}
```

代码清单15-11没有什么实际意义，但我们只关注流控制。在开始之前，有必要指出以下几点。

- 该方法包含一个参数（text）。
- 该方法包含一个循环，后续操作执行时需跳回该循环内。
- 该方法包含两个不同类型的await表达式：Task.Delay返回一个Task，而Task.Yield()则返回一个YieldAwaitable。
- 该方法包含显式的局部变量（total、ch和unicode），需在不同的调用间关注其变化。
- 该方法包含一个通过调用text.GetEnumerator()方法创建的隐式局部变量。
- 该方法最终返回一个值。

这段代码最初的版本将text作为string类型的参数，但C#编译器会对字符串的迭代进行优化，并使用Length属性和索引器，这会使反编译后的代码变得更加复杂。

我不会给出完整的反编译后代码，不过大家可在下载资源中自行下载此段代码。在接下来的几节中，我们将介绍一些最重要的部分。如自行反编译，则不会看到完全相同的代码；我对变量和类型进行了重命名，从而使其在功能不变的前提下更加易读。

我们先从最简单的部分——骨架方法——开始。

15.5.2　骨架方法的结构

尽管骨架方法中的代码非常简单，但它暗示了状态机的职责。代码清单15-11生成的骨架方法如下所示：

```
[DebuggerStepThrough]
[AsyncStateMachine(typeof(DemoStateMachine))]
static Task<int> SumCharactersAsync(IEnumerable<char> text)
{
    var machine = new DemoStateMachine();
    machine.text = text;
    machine.builder = AsyncTaskMethodBuilder<int>.Create();
    machine.state = -1;
    machine.builder.Start(ref machine);
    return machine.builder.Task;
}
```

AsyncStateMachineAttribute类型是为async引入的新特性（attribute）之一。它是为工具而设计的，你自己并不会有机会消费这个特性，并且也不应该在自己的方法上应用这个特性。

我们已经在这个状态机上看到了三个字段。

- 一个是参数（text）。显然有多少个参数就会有多少个字段。
- 一个是AsyncTaskMethodBuilder<int>。该结构负责将状态机和骨架方法联系在一起。对于仅返回Task的方法，存在对应的非泛型类。对于返回void的方法，可以使用AsnycVoidMethodBuilder结构。

- 一个是`state`，值从`-1`开始。初始值永远为`-1`，稍后我们会介绍其他值的含义。

由于状态机是一个结构（struct），`AsyncTaskMethodBuilder<int>`也是一个结构，因此我们还没有执行任何堆分配。当然，完全可以让执行的不同调用在堆上进行分配，但有必要指出的是，代码在尽可能地避免这么做。异步的本质意味着，如果哪个`await`表达式需要真正的等待，你会需要很多这种值（在堆上），但代码保证了它们只会在需要的时候进行装箱。所有这些都属于实现细节，就像堆和栈属于实现细节一样，但为了让`async`能够适用于尽可能多的场景，微软的相关团队紧密合作，将分配降低到了绝对最小值。

对`machine.builder.Start(ref machine)`的调用非常有意思。这里使用了按引用传递，以避免创建状态机的复本（以及builder的复本），这是出于性能和正确性两方面的考虑。编译器非常愿意将状态机和builder视为类，因此`ref`可以在代码中自由地使用。为了使用接口，不同的方法将builder（或awaiter）作为参数，使用泛型类型参数，并限定其实现某个接口（如对于状态机来说就是`IAsyncStateMachine`）。这样在调用接口的成员时，就不需要任何装箱了。方法的行为描述起来非常简单——它让状态机同步地执行第一个步骤，并在方法完成时或到达需等待的异步操作点时得以返回。

第一个步骤完成后，骨架方法将返回builder中的任务。状态机在结束时，会使用builder来设置结果或异常。

15.5.3 状态机的结构

状态机的整体结构非常简单。它总是使用显式接口实现，以实现.NET 4.5引入的`IAsyncStateMachine`接口，并且只包含该接口声明的两个方法，即`MoveNext`和`SetStateMachine`。此外，它还拥有大量私有或公共字段。

状态机的声明在折叠后如代码清单15-11所示：

```
[CompilerGenerated]
private struct DemoStateMachine : IAsyncStateMachine
{
   public IEnumerable<char> text;

   public IEnumerator<char> iterator;
   public char ch;
   public int total;
   public int unicode;

   private TaskAwaiter taskAwaiter;
   private YieldAwaitable.YieldAwaiter yieldAwaiter;
   public int state;
   public AsyncTaskMethodBuilder<int> builder;
   private object stack;

   void IAsyncStateMachine.MoveNext() { ... }

   [DebuggerHidden]
   void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine machine)
   { ... }
}
```

❶ 为参数设计的字段

❷ 为局部变量设计的字段

❸ 为awaiter设计的字段

❹ 通用的基础设施

在这段代码中，我将字段分割为不同的部分。我们已经知道表示原始参数的text字段❶是由骨架方法设置的，而builder和state字段亦是如此，三者皆是所有状态机共享的通用基础设施。

由于需在多次调用MoveNext()方法时保存变量的值，因此每个局部变量也同样拥有着自己的字段❷。有时局部变量只在两个特殊的await表达式之间使用，而无须保存在字段中，但就我的经验来说，当前实现总是会将它们提升为字段。此外，这么做还可以改善调试体验，即使没有代码再使用它们，也无须担心局部变量丢值了。

异步方法中使用的awaiter如果是值类型，则每个类型都会有一个字段与之对应，而如果是引用类型（编译时的类型），则所有awaiter共享一个字段。本例有两个await表达式，分别使用两个不同的awaiter结构类型，因此有两个字段❸。如果第二个await表达式也使用了一个TaskAwaiter，或者如果TaskAwiater和YieldAwiter都是类，则只会有一个字段。由于一次只能存活一个awaiter，因此即使一次只能存储一个值也没关系。我们需要在多个await表达式之间传播awaiter，这样就可以在操作完成时得到结果。

有关通用的基础设施字段❹，我们已经了解了其中的state和builder。state用于跟踪踪迹，这样后续操作可回到代码中正确的位置。builder具有很多功能，包括创建骨架方法返回的Task和Task<T>，即异步方法结束时传播的任务，其内包含有正确结果。stack字段略微有点晦涩。当await表达式作为语句的一部分出现，并需要跟踪一些额外的状态，而这些状态又没有表示为普通的局部变量时，才会用到stack字段。15.5.6节将介绍一个相关示例，该示例不会用于代码清单15-11生成的状态机中。

编译器的所有魔法都体现在MoveNext()方法中，但在介绍它之前，我们先来快速浏览一下SetStateMachine。在每个状态机中，它都具有完全相同的实现，如下所示：

```
void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine machine)
{
   builder.SetStateMachine(machine);
}
```

简单来说，该方法的作用是：在builder内部，让一个已装箱状态机的复本保留有对自身的引用。我不想深入介绍如何管理所有的装箱，你只需了解状态机可在必要时得到装箱，同时，异步机制的各方面可保证在装箱后，还会一直使用这个已装箱的复本。这非常重要，因为我们使用的是可变值类型（不寒而栗！）。如果允许对状态机的不同复本进行不同的修改，那么整个程序很快就会崩溃。

换一个角度来说（如果你开始认真思考状态机的实例变量是如何传播的，这就会变得很重要），状态机之所以设计为struct，就是为了避免早期不必要的堆分配，但大多数代码都将其视作一个类。围绕SetStateMachine的那些引用，让这一切正常运作。

现在万事俱备，只欠异步方法中的实际代码了。下面我们来研究MoveNext()。

15.5.4　一个入口搞定一切

如果你反编译过异步方法（我非常希望你会这么做），会看到状态机中的MoveNext()方法非常长，变化非常快，像是一个计算有多少await表达式的函数。它包含原始方法中的所有逻辑，

和处理所有状态变换所需要的芭蕾舞步[①]，以及用来处理整个结果或异常的包装代码。

在手动编写异步代码时，你通常会将后续操作分散到多个方法内：在一个方法内开始，然后在另一个方法内继续，并且可能在第三个方法中结束。但这样很难处理循环等流控制，对C#编译器来说更是不可能的。这和生成代码的可读性差是两码事。状态机具有单独的入口，即MoveNext()方法。该方法在一开始便投入使用，并且可用于所有await表达式的后续操作。每当调用MoveNext()方法时，状态机就会通过state字段计算出方法要跳转到的位置。在准备计算结果时，则跳转到方法的逻辑起始位置或await表达式的末尾。每个状态机只执行一次操作。实际上，在方法内部存在一个基于state的switch语句，每种情况都具有包含不同标签的对应goto语句。

MoveNext()方法一般为以下形式：

```
void IStateMachine.MoveNext()
{
   // 对于声明返回Task<int>的异步方法
   int result;
   try
   {
      bool doFinallyBodies = true;
      switch (state)
      {
         // 跳转到正确的位置
      }

      // 方法的主体
   }
   catch (Exception e)
   {
      state = -2;
      builder.SetException(e);
      return;
   }
   state = -2;
   builder.SetResult(result);
}
```

初始状态始终为-1，方法执行时状态也是-1（与等待时被暂停相反）。非负值均表示一个后续操作的目标。状态机在结束时状态为-2。在调试配置下创建的状态机中，你会看到一个指向-3状态的引用——此状态是我们未曾预料到的。退化的switch语句会导致糟糕的调试体验，而-3状态即是为避免该退化语句的出现而存在的。

在方法执行过程中，在原始异步方法的return语句处，会设置result变量。然而在到达方法的逻辑末尾时，将其用于builder.SetResult()的调用。即使是非泛型的AysncTaskMethodBuilder和AsyncVoidMethodBuilder类型，也包含SetResult()方法。前者表示对于从骨架方法返回的任务来说，该方法已经完成；后者则表示原始的SynchronizationContext已经完成。（异常会以同样的方式向原始的SynchronizationContext传播。这是一种相当丑陋的跟踪方式，但却对必须使用void方法的场景提供了一种解决方案。）

① 它真的很像是一场舞蹈，具有在正确时间和地点执行的复杂步骤（舞步）。

doFinallyBodies变量用于计算执行过程离开try块的作用域时，原始代码中的finally块（包括using或foreach语句中的隐式finally块）是否应该执行。理论上，只有以正常方式离开try块的作用域时，我们才希望执行finally块。如果我们只是从之前为awaiter附加了后续操作的方法中返回，由于该方法逻辑上已经“暂停”了，因此我们不希望再执行finally块。finally块均位于相关的try块之后，并出现在代码方法部分的Main主体中。

从原始异步方法的角度来看，大多方法体都是可识别的。当然，你需要习惯于所有局部变量都作为状态机的实例变量，但这并不是什么难事。正如你所想的那样，棘手之处是await表达式。

15.5.5　围绕**await**表达式的控制

任何await表达式均表示执行路径的一个分支。首先，被等待的异步操作得到一个awaiter，然后检查其IsCompleted属性。若返回true，即可立即获得结果并继续。否则，需进行以下处理。

- 存储awaiter，以供后面使用。
- 更新状态，以表示从哪里继续。
- 为awaiter附加后续操作。
- 从MoveNext()返回，确保不会执行任何finally块。

然后，在调用后续操作时，需跳转到正确的地方，获取awaiter并重置状态，然后继续。

例如，代码清单15-11中的第一个await表达式即：

```
await Task.Delay(unicode);
```

所生成的代码如下所示：

```
TaskAwaiter localTaskAwaiter = Task.Delay(unicode).GetAwaiter();
   if (localTaskAwaiter.IsCompleted)
   {
      goto DemoAwaitCompletion;
   }
   state = 0;
   taskAwaiter = localTaskAwaiter;
   builder.AwaitUnsafeOnCompleted(ref localTaskAwaiter, ref this);
   doFinallyBodies = false;
   return;
DemoAwaitContinuation:
   localTaskAwaiter = taskAwaiter;
   taskAwaiter = default(TaskAwaiter);
   state = -1;
DemoAwaitCompletion:
   localTaskAwaiter.GetResult();
   localTaskAwaiter = default(TaskAwaiter);
```

如果等待的操作有返回值（如使用HttpClient分配await client.GetStringAsync(...)的结果），那么上述代码末尾处的GetResult()调用将得到该值。

AwaitUnsafeOnCompleted方法将后续操作附加给awaiter，MoveNext()方法开头的switch语句可确保再次执行MoveNext()时，将控制传递给DemoAwaitContinuation。

> 说明 **AwaitOnCompleted和AwaitUnsafeOnCompleted** 在此前展示的一组接口中，IAwaiter<T>扩展了INotifyCompletion及其OnCompleted方法，此外还扩展了ICriticalNotifyCompletion接口及其UnsafeOnCompleted方法。状态机为实现ICriticalNotifyCompletion的awaiter调用builder.AwaitUnsafeOnCompleted，或为只实现INotifyCompletion的awaiter调用builder.AwaitOnCompleted。15.6.4节在讨论可等待模式如何与上下文交互时，会介绍这两个调用间的区别。

注意，编译器为awaiter消除了局部变量和实例变量，这样就可以适时进行垃圾回收。

如果单个的await表达式也可以像这样找到块，则生成代码在反编译模式下不会太难以阅读。由于CLR的限制，可能会存在较多的goto语句（及相应的标签），但在我看来，await模式才是最难理解的。

还有一个概念必须加以解释，那就是状态机中神秘的stack变量。

15.5.6 跟踪栈

谈到栈帧（stack frame）时，可能会想到在方法中声明的局部变量。当然，可能还会注意到一些隐藏的局部变量，如foreach循环中的迭代器。但栈上的内容不止这些，至少逻辑上是这样[①]。很多情况下，在一些表达式还没有计算出来前，另一些中间表达式是不能使用的。最简单的例子莫过于加法等二进制操作和方法调用了。

举个极简单的例子，思考下面这一行：

```
var x = y * z;
```

在基于栈的伪代码中，将为如下形式：

```
push y
push z
multiply
store x
```

现在假设有如下await表达式：

```
var x = y * await z;
```

在等待z之前，需计算y并将其保存至某处，但可能会从MoveNext()方法立即返回，因此需要一个逻辑栈来存储y。在执行后续操作时，可以重新存储该值，然后执行乘法。在这种情况下，编译器可将y的值赋值给stack实例变量。这会引起装箱，但同时也意味着可以使用单个变量。

这是个简单的例子。假设有多个值需要存储，如下所示：

```
Console.WriteLine("{0}: {1}", x, await task);
```

在逻辑栈上需要存储格式化的字符串和x值。此时编译器会创建一个包含两个值的

① 就像Eric Lippert常挂在嘴边的：栈是实现细节。有些变量你认为会在栈上，但实际在堆上，而有些变量可能仅存在于寄存器中。本节只讨论逻辑上会在栈上发生的事情。

Tuple<string, int>，并将其存储在stack的引用上。和awaiter一样，同一时间只需要一个逻辑栈，因此一直使用相同的变量是没有问题的[①]。在后续操作中，可以从元组（tuple）中获取实参，并用于方法调用。可下载的源代码中包含了完整的反编译示例，其中包括以上两条语句（LogicalStack.cs和LogicalStackDecompiled.cs）。

第二条语句最终将使用以下代码：

```
    string localArg0 = "{0} {1}";
    int localArg1 = x;
    localAwaiter = task.GetAwaiter();
    if (localAwaiter.IsCompleted)
    {
       goto SecondAwaitCompletion;
    }
    var localTuple = new Tuple<string, int>(localArg0, localArg1);
    stack = localTuple;
    state = 1;
    awaiter = localAwaiter;
    builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
    doFinallyBodies = false;
    return;
SecondAwaitContinuation:
    localTuple = (Tuple<string, int>) stack;
    localArg0 = localTuple.Item1;
    localArg1 = localTuple.Item2;
    stack = null;
    localAwaiter = awaiter;
    awaiter = default(TaskAwaiter<int>);
    state = -1;
SecondAwaitCompletion:
    int localArg2 = localAwaiter.GetResult();
    Console.WriteLine(localArg0, localArg1, localArg2);
```

此处加粗显示的是与逻辑栈元素相关的代码。

目前所有需了解的内容均已介绍完毕。如果你跟上了我们的步伐，就肯定比99%的开发者更为了解背后的细节。第一次没能完全理解也没有关系。在阅读这些状态机代码时，如果感到无法理清头绪，可以暂时放松一下，过一会儿再来继续学习。

15.5.7　更多内容

想了解更多细节内容吗？请打开一个反编译器。建议打开一个非常小的程序来研究编译器做了什么。如果程序很大，则很容易迷失在大量曲折且相似的后续操作中。你需要降低反编译器的优化程度，以得到相对低级的代码，而不是某种翻译。毕竟，一个完美的反编译器会重新生成异步函数，而这违背了我们练习的初衷。

① 不可否认，很多时候对于变量的类型，编译器可以更智能，或避免包含不必要的变量，但这一切可能要等到以后的版本再做优化了。

编译器生成的代码无法一直反编译成有效的C#。因为总是会出现故意对变量和类型使用不可读名称的问题，更重要的是，有时有效的IL并没有直接对应的C#。例如，在IL中跳转到循环指令是合法的，这是因为IL并没有循环的概念。而C#则无法从循环外部`goto`到循环内部的标签，因此很难完全正确地表示此类指令。C#编译器甚至不能完全使用自己的方式：IL在跳转目标时也有一些限制，因此常常会看到编译器不得不遍历一系列跳转，从而找到正确的位置。

同样，有些反编译器在处理逻辑栈附近赋值语句的确切顺序时仍有些混乱，偶尔会将临时变量（如`localArg0`和`localArg1`）的赋值语句，错误地放到检查`IsCompleted`的地方。这是因为代码与C#编译器正常生成的代码不尽相同。在得知寻找目标后，情况则不算特别糟糕，但这意味着有时需直接面对IL。

15.6 高效地使用 async/await

了解异步函数的行为及其背后的原理后，就可以成为一名异步编程专家了，对吗？显然不是[①]。与编程的许多方面一样，经验是最重要的。而目前，很少有人对异步函数有着丰富的经验。尽管我不能给你经验，但可以提供一些让工作更加轻松的提示和技巧。

撰写本书之时，对C# 5异步编程最熟悉的人全都在微软，他们在开发时与C# 5异步编程朝夕相处，并接收来自beta版试用者的反馈，等等。因此，我强烈推荐访问并行编程团队的博客（http://blogs.msdn.com/b/pfxteam/），以了解更多建议内容。

当然，本书内容也很丰富……

15.6.1 基于任务的异步模式

C# 5异步函数特性的一大好处是，它为异步提供了一致的方案。但如果在命名异步方法以及触发异常等方面做法存在着差异，则很容易破坏这种一致性。微软因此发布了基于任务的异步模式（Task-based Asynchronous Pattern，TAP），即提出了每个人都应遵守的约定。TAP有单独的文件（http://mng.bz/B68W），MSDN中也有它的页面（http://mng.bz/4N39）。

微软当然也遵循了这些约定。.NET 4.5包含了适用于所有场景的大量异步API。与命名、类型设计等遵循普通的.NET约定一样，如异步代码也遵循同样的约定，其他开发者会觉得这些代码非常容易阅读和使用。

TAP相当容易理解，而且只占38页的篇幅。强烈建议阅读完整的文档。本节后续部分将介绍我认为最重要的那些部分。

异步方法的名称应以`Async`为后缀，如`GetAuthenticationTokenAsync`、`FetchUserPortfolioAsync`等。而这已经在.NET Framework中引起了一些冲突，如`WebClient`中就包含`DownloadStringAsync`等遵循基于事件异步模式的方法，因此基于TAP的新方法，其名称看上去有点丑陋，如`DownloadStringTaskAsync`、`DownloadDataTaskAsync`等。如果自己的代码

15

① 当然，你可能已经是一名异步编程专家了，但仅仅阅读本章内容是不能让你达到这个高度的。

中也存在这种命名冲突，建议使用TaskAsync后缀。如果方法很明显是异步的，则可去掉后缀，如Task.Delay和Task.WhenAll等。一般来说，如果方法的整个业务是异步的，而不是为了达到某种业务上的目标，那么去掉后缀应该就是安全的。

TAP方法一般返回的是Task或Task<T>，但也有例外，如可等待模式的入口Task.Yield，不过这实属凤毛麟角。重要的是，从TAP方法中返回的任务应该是“热”的。也就是说，它表示的操作应该已经开始执行了，而无须调用者的手动开启。对大多数开发者来说，这是显而易见的，但某些平台会创建“冷”任务，只有在显式地请求后才会完成启动，这有点类似于C#中的迭代器块。特别是F#也遵循这一约定，而且在响应式扩展（Rx）中，也需要考虑到这一点。

创建异步方法时，通常应考虑提供4个重载。4个重载均具有相同的基本参数，但要提供不同的选项，以用于进度报告和取消操作。假设要开发一个异步方法，其逻辑与下列同步方法相同：

```
Employee LoadEmployeeById(string id)
```

根据TAP的约定，需提供下列重载的一个或全部：

```
Task<Employee> LoadEmployeeById(string id)
Task<Employee> LoadEmployeeById(string id, CancellationToken cancellationToken)
Task<Employee> LoadEmployeeById(string id, IProgress<int> progress)
Task<Employee> LoadEmployeeById(string id,
    CancellationToken cancellationToken, IProgress<int> progress)
```

这里的IProgress<int>是一个IProgress<T>，这里的T可以是任何适用于进度报告的类型。例如，倘若异步方法要逐一处理一些记录，则可使用IProgress<Tuple<int, int>>，从而报告已处理的记录数和总的记录数。

我没有将进度报告硬当成是操作，因为这真的没有意义。取消操作通常来说更容易支持，因为存在着很多框架方法的支持。如果异步方法主要是执行其他异步操作（可能还包括依赖关系），就很容易支持取消操作，只需接收一个取消token并向下游传递即可。

异步操作应同步地进行错误检查，如不合法的实参等。这有点笨拙，但通过15.3.6节的分离方法或15.4节的在单个方法中使用匿名异步函数，可以很容易地实现这一操作。尽管延迟验证参数看上去很诱人，但却很难定位难以诊断的失败。

基于IO的操作会将工作移交给硬盘或其他计算机，这非常适合异步，而且没有明显的缺点。CPU密集型的任务就不那么适合了。可以很轻松地将一些工作移交给线程池，在.NET 4.5中也要比以前更加简单，这都要感谢Task.Run方法，但使用代码库来实现这一点，相当于为调用者做了决定。不同的调用者可能会有不同的需求。如果仅仅暴露同步风格的方法，相当于为调用者提供了灵活性，以保证其以最适合的方式进行工作。它们可以在需要时开始一个新任务，如果可以接受当前线程在一段时间内忙于执行方法，也可以实现同步调用。

如果任务需等待其他系统返回的结果，而随后的结果处理又十分耗时，这种情况就更加棘手了。尽管我认为严格的指南不会有太大帮助，但在文档中指明这种行为还是非常重要的。如果最终要占用调用者上下文的大部分CPU资源，就应该清晰地进行说明。

另一种方法是避免使用调用者的上下文，而应使用Task.ConfigureAwait方法。该方法目

前只包含一个continueOnCapturedContext参数，但为了清晰，应使用命名参数来进行指定。该方法返回一个可等待模式的实现。当参数为true时，可等待的行为正常，因此如果UI线程调用异步方法，await表达式后面的后续操作可仍然在UI线程上执行。这样要访问UI元素就变得非常方便。如果没有任何特殊需求，可将参数指定为false，这时后续操作的执行通常发生在原始操作完成的上下文中①。

对于一个混合操作（获取数据、处理数据、将数据存储到数据库）来说，代码可能如下所示：

```
public static async Task<int> ProcessRecords()
{
   List<Record> records = await FetchRecordsAsync()
      .ConfigureAwait(continueOnCapturedContext: false);

   // 记录处理
   await SaveResultsAsync(results)
      .ConfigureAwait(continueOnCapturedContext: false);

   // 让调用者知道处理了多少条记录
   return records.Count;
}
```

该方法大部分代码在线程池线程上执行。由于并没有要求在原始线程中执行，因此这正是我们想要的结果。（专业术语叫做该操作没有线程亲和力（thread affinity）。）但这并不会影响调用者，如果异步UI方法等待的是调用ProcessRecords的结果，则该异步方法将继续在UI线程上执行。只有在ProcessRecords内部的代码声明其不关心执行上下文时，才会在线程池线程上执行。

有争议的是，没有必要在第二个await表达式处调用ConfigureAwait，因为所剩工作已经不多了，但通常来说我们应该在每个await表达式处调用该方法，而保持一致是个好习惯。如果想为调用者提供方法执行上下文的灵活性，可将其作为异步方法参数。

注意，ConfigureAwait只会影响执行上下文的同步部分。而有关模拟（impersonation）等其他部分，传播则并不关心，15.6.4节将详细介绍相关内容。

说明　**TPL数据流**　尽管TAP只是一些约定和示例，但微软还是建立了一个单独的库，即“TPL数据流”，从而为一些特殊场景（特别是那些可以通过生产者/消费者模式建模的场景）提供高级构建块。可以通过NuGet包（Microsoft.Tpl.Dataflow）来获取这个库。它是免费的，并且含有大量指导文件。即使不直接使用，也有必要看一看，来感受一下如何设计并行程序。

即使没有额外的库，也能在遵循常规设计原则的前提下，构建优雅的异步代码。组合便是这些重要的设计原则之一。

① “通常”并不等于“绝对”。尽管文档中没有指明细节，但有时我们确实不希望它们在相同的上下文中执行。应将ConfigureAwait(false)当成是 “我不在乎后续操作在哪儿执行”，而不是显式地将其附加到指定上下文中。

15.6.2　组合异步操作

对于C# 5异步特性，我最喜欢的一点是它可以自然而然地组合在一起。这表现为两种不同的方式。最明显的是，异步方法返回任务，并通常会调用其他返回任务的方法。这些方法可以是直接的异步操作（如链的最底部），也可以是更多的异步方法。所有的包装和拆包都需要将结果转换为任务，反向操作则由编译器完成。

另一种组合形式是，创建与操作无关的构建块来管理任务的处理。这些构建块无须知道任务在做什么，而只是单纯待在`Task<T>`的抽象级别。这有点像LINQ操作符，只是面向的是任务而不是序列。框架中内置了一些构建块，但也可以自行创建。

1. 在单个调用中收集结果

例如，尝试获取若干URL。15.3.6节中一次性获取了所有URL，并在完成任务后立即停止获取。假设这次要启动多个并行请求，然后每得到一个URL就记录下结果。记住，异步方法返回的是已经运行的任务，因此可非常轻松地为每个URL启动一个任务：

```
var tasks = urls.Select(async url =>
{
   using (var client = new HttpClient())
   {
      return await client.GetStringAsync(url);
   }
}).ToList();
```

注意，需调用`ToList()`来具体化LINQ查询。这保证了每个任务将只启动一次。否则每次迭代`tasks`时，将会再次获取字符串。（如不释放`HttpClient`，代码会更加简单，但即便如此，代码也不是很难看。）

TPL提供了一个`Task.WhenAll`方法，从而将各有一个结果的多个任务组合成一个包含多个结果的任务。常用的方法重载签名如下所示：

```
static Task<TResult[]> WhenAll<TResult>(IEnumerable<Task<TResult>> tasks)
```

这个声明看上去非常糟糕，但在真正使用时，会发现其方法目的非常单纯。你将得到一个`List<Task<string>>`，因此可以写为：

```
string[] results = await Task.WhenAll(tasks);
```

所有任务均已结束，并将结果收集到一个数组中后，等待方可终止。本章前面讲过，如果多个任务抛出异常，则只有第一个异常会立即抛出，但可总是迭代这些任务，以找到具体失败的任务及其失败原因，或使用代码清单15-2中所示的`WithAggregatedException`扩展方法。

如果只关注第一个返回的请求，则可使用`Task.WhenAny`方法。该方法不会等待第一个成功完成的任务，而只会等待第一个到达终点状态的任务。

本例中，你可能想要点特别的做法。在任务完成后报告全部结果可能会更有用些。

2. 在全部完成时收集结果

`Task.WhenAll`是.NET内置的转换构建块（transformational building block），接下来将介绍如

何以类似的方式构建自己的方法。TAP文档中含有类似的示例代码，从而创建了Interleaved方法，这里将介绍另一版本。

代码清单15-12旨在传递一个输入任务的序列，并返回一个输出任务的序列。两个序列中任务的结果是相同的，但存在一个重要差异，即输出任务的完成顺序与输入完全一样，因此可以一次await一个任务，并可立即得到任务结果。这听上去有些神奇，对我来说也是如此，因此我们来看看代码，研究一下它的工作原理。

代码清单15-12　按完成顺序将任务序列转换到新的集合

```
public static IEnumerable<Task<T>> InCompletionOrder<T>
    (this IEnumerable<Task<T>> source)
{
   var inputs = source.ToList();
   var boxes = inputs.Select(x => new TaskCompletionSource<T>())
                     .ToList();

   int currentIndex = -1;
   foreach (var task in inputs)
   {
      task.ContinueWith(completed =>
      {
         var nextBox = boxes[Interlocked.Increment(ref currentIndex)];
         PropagateResult(completed, nextBox);
      }, TaskContinuationOptions.ExecuteSynchronously);
   }
   return boxes.Select(box => box.Task);
}
```

代码清单15-12依赖TPL中一个非常重要的类型，即TaskCompletionSource<T>。该类型可用于创建一个尚未含有结果的Task，并在之后提供结果（或异常）。它和AsyncTaskMethodBuilder<T>都建立在相同的基础结构之上。后者为异步方法提供返回的Task，并在方法体完成时，将带结果的任务向外传播。

为什么会用这么奇怪的变量名（boxes）呢？我常常把任务想象成纸箱，这些纸箱承诺（promise）在某个时刻，其内部会含有值或错误。TaskCompletionSource<T>就像是背面有洞的箱子，你可以把它给别人，然后再偷偷地把值从洞口塞进去[①]。这正是PropagateResult方法的作用，不过它没那么有意思，所以此处不予列出，基本上它会将已完成的Task<T>的结果传播到TaskCompletionSource<T>中。如果原始任务正常完成，则将返回值复制到TaskCompletionSource<T>中。如果原始任务产生了错误，则可将异常复制到TaskCompletionSource<T>中。取消原始任务后，TaskCompletionSource<T>也会随之被取消。

真正聪明的部分是（我对此说法不承担任何责任——有人发邮件建议我加入这一免责声明），在该方法运行时，它并不知道哪个TaskCompletionSource<T>会对应哪个输入任务，而只是将相同的后续操作附加到各任务上，然后由后续操作来寻找下一个TaskCompletionSource<T>（通过对一个计数器进行原子地累加）并传播结果。也就是说，它会按照原始任务的输出顺序对

15

① 如果这与量子物理学雷同，那纯属巧合。此外，如若有人用Task<Cat>做实验，本人对因此造成的后果概不负责。

箱子进行填充。

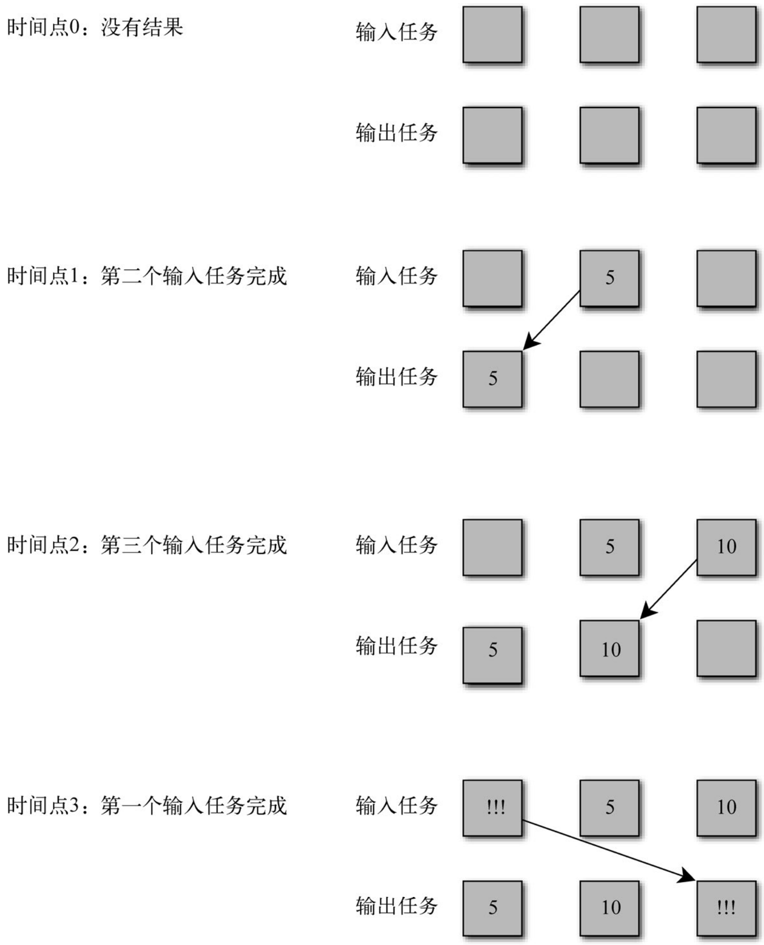


图15-5　输入任务和输出任务的顺序

图15-5展示了三个输入任务，以及相应的由方法返回的输出任务。即使输入任务的顺序与方法返回的顺序不同，输出任务的顺序也会与之相同。

有了这个绝妙的扩展方法后，即可编写代码清单15-13，从而得到一组URL，并行地对每个URL发起请求，并在请求完成时写下各页面的长度，然后返回总长度。

代码清单15-13　在数据返回时显示页面长度

```
static async Task<int> ShowPageLengthsAsync(params string[] urls)
{
   var tasks = urls.Select(async url =>
   {
      using (var client = new HttpClient())
      {
```

```
            return await client.GetStringAsync(url);
        }
    }).ToList();

    int total = 0;
    foreach (var task in tasks.InCompletionOrder())
    {
        string page = await task;
        Console.WriteLine("Got page length {0}", page.Length);
        total += page.Length;
    }
    return total;
}
```

代码清单15-13存在两个小问题。

- 一个任务失败，则整个异步操作都将失败，并且不会保留结果。这也许没问题，但也可能希望能够将每次失败记录下来。（与.NET 4不同，不处理任务异常，则默认不会让进程当掉，但至少应考虑对其他任务产生的影响。）
- 失去对页面转向具体URL的跟踪。

这两个问题都可以通过少量代码轻松解决，也可以进一步提取成可复用的构建块。举这些例子并不是为了满足个别需求，而是为了让你接受组合带来的各种可能性。

TAP白皮书中并不是只有`Interleaved`这一个例子，它还包括很多概念，并附带一些有助于理解的示例。

15.6.3 对异步代码编写单元测试

我在编写本节时还有点诚惶诚恐。目前，我认为社区还没有足够的经验，来对如何检测异步代码给出确切的答案。毫无疑问，我们会走一些弯路，并得出几个自相矛盾的方案。重点在于，和同步代码一样，如果从一开始就考虑到代码的可测试性，即可有效地为异步代码编写单元测试。

1. 安全地注入异步

本节将展示一个针对场景的方案。在该场景中，可控制异步代码所依赖的异步操作。我并不是要说，如果代码使用了`HttpClient`和其他难以伪造的类型，编写测试有多么困难。此种情况时常发生，如果依赖于那些在测试中难以使用的类型，即可经常遇到类似的问题。

我们来测试上一节中介绍的“神奇排序”代码。假设要创建一些以特定顺序完成的任务，并且（至少在部分测试中）确保可以在两个任务的完成之间执行断言。此外，我们不想引入其他线程，而希望拥有尽可能多的控制和可预见性。实质上，我们希望能够控制时间。

我的解决方案是使用`TimeMachine`类来伪造时间，它可以用在特定时间以特殊方式完成的计划任务，以编程方式来推进时间。将其与Windows Forms消息泵的手工版本`SynchronizationContext`组合，可得到一个非常合理的测试框架（test harness）[①]。此处不会展示全部框架代码，因为它又

15

① 这里将test harness翻译为“测试框架”，除此之外本书其他地方出现的“测试框架”均为“test framework”。有关test harness的内容，请参考http://zh.wikipedia.org/wiki/Test_harness。——译者注

长又无趣，不过可从示例代码中找到它。接下来为一些测试示例。

先从全部成功的测试开始。如果让三个任务以1、2、3的顺序完成，然后将这三个任务按不同顺序重新组合，并调用InCompletionOrder，则所得结果的顺序不会发生改变：

```
[TestMethod]
public void TasksCompleteInOrder()
{
   var tardis = new TimeMachine();
   var task1 = tardis.ScheduleSuccess(1, "t1");
   var task2 = tardis.ScheduleSuccess(2, "t2");
   var task3 = tardis.ScheduleSuccess(3, "t3");

   var tasksOutOfOrder = new[] { task2, task3, task1 };

   tardis.ExecuteInContext(advancer =>
   {
      var inOrder = tasksOutOfOrder.InCompletionOrder().ToList();
      advancer.AdvanceTo(3);
      Assert.AreEqual("t1", inOrder[0].Result);
      Assert.AreEqual("t2", inOrder[1].Result);
      Assert.AreEqual("t3", inOrder[2].Result);
   });
}
```

ExecuteInContext方法临时性地将当前线程的SynchronizationContext替换为ManuallyPumpedSynchronizationContext（示例代码中也是如此），然后为方法参数指定的委托提供一个推进器。该推进器可以按特定数量推进时间点，以保证在适当的时间点完成任务（并执行后续操作）。本例只是简单地“快进”，直到任务全部完成。

下列测试演示了如何以更加细粒度的方式来控制时间：

```
// 准备部分与上例相同，因此没有列出
tardis.ExecuteInContext(advancer =>
{
   var inOrder = tasksOutOfOrder.InCompletionOrder().ToList();

   Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[0].Status);
   Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[1].Status);
   Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

   advancer.Advance();
   Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[0].Status);
   Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[1].Status);
   Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

   advancer.Advance();
   Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[1].Status);
   Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

   advancer.Advance();
   Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[2].Status);
});
```

可以看到，任务按照正确的顺序完成。

为什么这里的时间都是整数，而不能是DateTime或TimeSpan呢？这是深思熟虑后的结果。

我们真正拥有的时间线是由TimeMachine建立的仿造时间线，真正感兴趣的是任务完成的时间点。

当然，被测方法在以下两方面略有不同。

- 没有真正用async实现。
- 直接以参数的形式提供任务。

如果测试的是更加专注于业务的异步方法，则要为依赖的任务设置所有的结果，推进时间以完成所有任务，然后检查返回任务的结果。需以正常方式提供伪造的产品代码。此处异步带来的唯一不同是，不再使用stub和mock来返回调用的直接结果，而是要求返回TimeMachine产生的任务。控制反转的所有优点仍然适用，只是需要某种方式来创建合适的任务。

这种想法显然不适用于所有情况，但至少可以让各位认识到，在不胡乱调用Thread.Sleep的情况下是可以对异步代码进行单元测试的，同时测试碎片化的风险一直存在。

2. 运行异步测试

上一节介绍的测试是完全同步运行的，测试本身并没有使用async或await。如果所有测试中均使用了TimeMachine类，那么这样做是合理的，但在其他情况下，可能会需要编写用async修饰的测试方法。

可按如下方式轻松完成该操作：

```
[Test] // NUnit的TestAttribute
public async void BadTestMethod()
{
   // 使用了await的代码
}
```

在任何测试框架下，它都能编译通过[①]，但其行为可能并不是我们所预期的。特别是所有测试都可能会并行启动，并且十分有可能在做断言前就“结束”了。

巧合的是，NUnit从2.6.2版开始支持异步测试，以上方法会以非常智能的方式运作。但如果在早期版本中运行，则测试开始后，测试运行器（test runner）在遇到第一个await后便会停止。方法后任何可能的失败，都会发送给测试的SynchronizationContext，但测试本身并不需要它。

对于支持异步测试的测试框架来说，最好让测试返回Task，如下所示：

```
[Test]
public async Task GoodTestMethod()
{
   // 使用了await的代码
}
```

这样，测试框架可更容易地得知测试完成的时间，并检查是否失败。还有一个好处就是，不支持异步测试的测试框架不会运行此类测试，而不是给出警告，这比错误地运行测试要好得多。在撰写本书之时，最新版的NUnit、xUnit和Visual Studio Unit Test Framework（俗称MS Test）都已支持异步测试，其他框架可能也是如此。在编写这样的测试之前，请检查要使用的框架和版本。

① 当然，不同的测试框架会对测试方法应用不同的特性。——译者注

此外还应注意死锁问题。与上一节使用TimeMachine的测试不同，你可能不想让所有后续操作都运行在单独的线程上，除非该线程像UI线程那样。有时我们控制所有相关任务，并使用单线程上下文。而有时则需更加小心，只要测试代码本身不是并行执行的，即可用多线程来触发后续操作。这样的单元测试会让人头疼，但如果用同样的框架进行功能测试、集成测试，甚至是产品级别的探索，则会希望测试里运行的是真正的任务，而不是由TimeMachine提供的伪造任务。

我相信，社区日后一定会开发出一些伟大的工具，以供测试更多的代码。自然流畅的异步代码定会成为未来代码的重要组成部分，我可不想在没有测试的情况下编写此类代码。对于异步的学习已接近尾声，但我之前承诺过，会再次介绍生成代码中非常有意思的AwaitUnsafeOnCompleted方法调用。

15.6.4　可等待模式的归来

15.3.3节中列出了一些虚构的接口，用来传达可等待模式的基本思想。尽管我解释过这并非完全正确，但还是有点含糊其辞。如果要实现可等待模式或研究反编译代码，可能需要知道相关细节内容，否则无须了解这里面的蹊跷。

之前提及的真正接口是INotifyCompletion，如下所示：

```
public interface INotifyCompletion
{
   void OnCompleted(Action continuation);
}
```

另一个扩展了上述接口，并且也位于System.Runtime.CompilerServices命名空间的接口是：

```
public interface ICriticalNotifyCompletion : INotifyCompletion
{
   void UnsafeOnCompleted(Action continuation);
}
```

这两个接口的核心都是上下文。本章多次提到SynchronizationContext，你可能也曾看到过它。它是一个能将调用封送到适当线程的同步上下文，而不管该线程是特定的线程池线程，还是单个的UI线程，或是其他线程。不过这并不是唯一相关的上下文，此外还存在有SecurityContext、LogicalCallContext、HostExecutionContext等大量上下文。它们的最上层结构是ExecutionContext。它是所有其他上下文的容器，也是本节将要关注的内容。

ExecutionContext会跨过await，这一点非常重要。在任务完成时，你不会希望只是因为忘了所模拟的用户而再次回到异步方法中。为传递上下文，需在附加后续操作时捕获它，然后在执行后续操作时还原它。这分别由ExecutionContext.Capture和ExecutionContext.Run方法负责实现。

有两段代码可执行这种捕获/还原操作，即awaiter和AsyncTaskMethodBuilder<T>类（及其兄弟类）。你可能希望只选择其中一种，并无须操心后续事宜。但还有一些其他因素在起作用。你很容易忘记将执行上下文传递给awaiter，因此应在方法构造器代码中也实现一次。另一方面，

任何使用awaiter的代码都能直接访问它，因此你不希望在使用编译器生成代码时，因信赖所有调用者而暴露可能的安全隐患——这表明编译器生成代码应该存在于awaiter代码中。但同样地，你也不希望捕获和还原上下文两次，这种重复是没必要的。如何解决这个矛盾呢？

我们已经看到了答案：使用两个具有细微差别的接口。如果要实现可等待模式，则必须由`OnCompleted`方法来传递执行上下文。如果实现的是`ICriticalNotifyCompletion`，则`UnsafeOnCompleted`方法不应传递执行上下文，而应标记上`[SecurityCritical]`特性，以阻止不信任的代码调用。当然，方法的builder是可信的，用它们来传递上下文，可保证部分可信的调用者仍能有效地使用awaiter，但准攻击者则无法规避上下文流。

本节异常简洁，我发现整个上下文的话题多少会让人感到迷惑，而且还有一些更复杂的东西此处并没有谈及。如需实现自己的awaiter，而又不想委托给已知的awaiter（可能无须这么做），可阅读Stephen Toub的博文“ExecutionContext vs SynchronizationContext”（http://mng.bz/Ye65），以了解更多细节。

15.6.5　在WinRT中执行异步操作

Windows 8在应用程序生态系统中引入了Windows Store和WinRT。我在附录C中介绍了WinRT的一些细节。WinRT是一种现代的、面向对象的非托管环境，在很多方面，可将其看成是一个新的Win32。WinRT删除了一些熟悉的.NET类型，即便是那些保留下来的，大多也都失去了阻塞与IO有关的调用权力。

我们已经看到，CLR中的类型一般通过`Task<T>`来公开异步操作，但这个类型在WinRT中是不存在的。取而代之的，是很多扩展`IAsyncInfo`的接口：

- `IAsyncAction`；
- `IAsyncActionWithProgress<TProgress>`；
- `IAsyncOperation<TResult>`；
- `IAsyncOperationWithProgress<TResult, TProgress>`。

`Action`类型与`Operation`类型的区别，类似于`Task`与`Task<T>`或`Action`与`Func`的区别：`Action`没有返回值，而`Operation`有返回值。`WithProgress`版本将进度报告构建到单个类型中，而不是在每个TAP中要求方法包含带`IProgress<T>`的重载。

这些接口的细节超出了本书范围，但介绍其相关细节的资源十分丰富。建议从Stephen Toub的Windows 8博文“深入WinRT和await”（http://mng.bz/F1TF）开始学习。

处理C# 5这些接口时，需注意以下几点。

- `GetAwaiter`扩展方法可以直接等待行为（action）和操作。
- `AsTask`扩展方法可以将行为或操作看作是任务，并通过`IProgress<T>`提供取消token和进度报告。
- `AsAsyncOperation`和`AsAsyncAction`扩展方法则正好相反，可将任务转换为对WinRT友好的包装器（wrapper）。

15

这些扩展方法都是由`System.Runtime.WindowsRuntime.dll`程序集中的`System.WindowsRuntimeSystemExtensions`类提供的。

我们再一次看到了可等待模式的价值。C#编译器不在乎是调用扩展方法还是实例方法来等待异步操作。扩展方法的调用只是另一种可等待模式罢了。大多数时候，我们都能够在其原生类型中离开该异步操作，并实现照常等待。对于更复杂的场景，我们仍可将WinRT异步操作视为熟悉的`Task<T>`，这种灵活性是非常值得称道的。

另一种在WinRT模型中运行异步代码的方式是，使用`System.Runtime.InteropServices.WindowsRuntime.AsyncInfo`类的`Run`方法。如需将`IAsyncOperation`（或`IAsyncAction`）传递给其他代码，使用此方法要比调用`Task.Run(...).AsAsyncOperation`更加清晰。

在编写WinRT应用程序时，异步是必不可少的。很多时候，平台根本不会提供为IO编写同步代码的机会。当然，也可以自己处理平台所做的一切工作，但使用C# 5的异步特性会让WinRT更加易用。在C#增加异步特性的同时发布WinRT，显然不是巧合。微软可没有打算在这个领域浅尝辄止。这是用C#编写Windows Store应用程序的正确方式。

15.7　小结

但愿本章复杂、深入的内容没有掩盖C# 5异步特性的优雅。以更加熟悉的执行模型来编写高效的异步代码，是一大进步。人们对该特性的充分理解将具有革命性意义。以我有关异步的演讲经验来看，很多开发者在第一次看到和使用该特性时会感到迷惑不解。这完全是可以理解的，但请不要因此而放弃。希望本章能够解答一些学习中遇到的问题，网上的大量文档也可在一定程度上有所助益，此外Stack Overflow上还有大批热心人随时负责答疑解惑。

说到其他资源，需要强调的一点是，我尽量在本章中涵盖了异步的语言层面，以和本书其余部分内容相符。除了这些语言特性外，还需了解更多异步开发的相关知识，希望各位可以阅读TPL的所有相关资料。即使还不能使用C# 5，使用.NET 4也可以通过`Task<T>`来处理异步操作。如想使用原生`Thread`方法，只需考虑TPL是否能提供一种更加高级的抽象，以更简单的方式达到相同目的即可。

总之，C# 5的异步功能十分强悍。但这并不是C# 5的全部内容，还有一些小的特性会在本书最后介绍。

第 16 章 C# 5附加特性和结束语

本章内容

- 捕获变量的变化
- 调用者信息特性
- 结束语

C# 2除主要特性外，还包含大量小而独立的特性。C# 3也存在一些小特性用于构建LINQ。连C# 4也有几个相对较小的特性值得深入展开。

然而C# 5除了异步以外，则几乎没有其他特性了。C# 5只存在两个微不足道的附属物而已。C#设计团队一直在特性的成本（如设计、实现、测试文档、开发者教学）和收益之间权衡。他们肯定有大量优秀的特性需求想要实现，之所以选择了这两个，想必是因为它们小得恰到好处。

第一个变化称不上是特性，只是对之前语言设计中的错误进行修复罢了……

16.1 foreach 循环中捕获变量的变化

5.5.5节中曾经警告说，在foreach循环内的匿名函数（通常为Lambda表达式）中捕获循环变量时要格外小心。代码清单16-1就展示了这样一个简单的示例，它看上去似乎会输出x、y、z。

代码清单16-1 使用捕获的迭代变量

```
string[] values = { "x", "y", "z" };
var actions = new List<Action>();

foreach (string value in values)
{
   actions.Add(() => Console.WriteLine(value));
}

foreach (Action action in actions)
{
   action();
}
```

在C# 3和C# 4中，以上代码实际上会打印出三个z。循环变量（value）可由Lambda表达式捕获，且名义上在循环的每次迭代中，只有一个变量“实例”的值发生了变化。全部三个委托都

将引用相同的变量，并且在最终执行时，该变量的值为z。这并不是编译器实现上的错误，而是语言被指定所产生的行为。

在C# 5中，语言的行为与当初预期的一样：循环的每次迭代都可有效地引入一个单独变量。每个委托都引用不同的变量，变量的值就是这次循环迭代中产生的值。

有关该特性的内容就讲到这里，它只是修复了会让很多开发者产生疑惑的语言部分而已。（Stack Overflow上有超多人询问这方面的问题。）

但此处要提醒一句：如果编写的代码需用不同版本的C#编译器进行编译，则应注意它们产生的行为是不同的。对于任何版本的C#来说，代码清单16-1都不会产生警告，而在C# 5中，行为却神不知鬼不觉地发生了改变。要慎之又慎，并且确保有单元测试可以依靠。

下面是最后一个特性……

16.2　调用者信息特性

有些特性是非常一般化的，如Lambda表达式、隐式类型局部变量，以及泛型等。有些特性则更为特殊，如LINQ就是为了查询某种形式的数据，尽管其目的是归纳多种不同的数据源。C# 5的这一特性（feature）是很有针对性的：有两种重要的使用场景（一个显而易见，另一个则没那么明显），而且我真的不希望会在其他情况下使用它们。

16.2.1　基本行为

.NET 4.5引入了三个新特性（attribute），即CallerFilePathAttribute、CallerLineNumberAttribute和CallerMemberNameAttribute。三者均位于System.Runtime.CompilerServices命名空间下。和其他特性一样，在应用时可以省略Attribute后缀。鉴于这是最常见的特性用法，本书后续内容会进行适当地缩写。

这三个特性都只能应用于参数，并且只有在应用于可选参数时才有用。其理念非常简单：如果调用点没有提供实参，则编译器可使用当前文件、行数或成员名来作为实参，而不使用常规的默认值。如果调用者提供了实参，编译器则将忽略这些特性。

代码清单16-2展示了这两种情况。

代码清单16-2　使用调用者信息特性

```
static void ShowInfo([CallerFilePath] string file = null,
                     [CallerLineNumber] int line = 0,
                     [CallerMemberName] string member = null)
{
   Console.WriteLine("{0}:{1} - {2}", file, line, member);
}
...
ShowInfo();                                     // 编译器填充所有参数
ShowInfo("LiesAndDamnedLies.java", -10);        // 编译器只填充name
```

代码清单16-2可输出以下内容：

```
c:\Users\Jon\Code\Chapter16\CallerInfoDemo.cs:21 - Main
LiesAndDamnedLies.java:-10 - Main
```

当然，并不需要总是为这些参数提供虚拟值，但显式传递还是很有用的，尤其是想使用同样的特性来记录当前方法调用者的时候。

成员名特型适用于所有成员[①]，但下列成员将使用特殊的名称：

- 静态构造函数：.cctor；
- 构造函数：.ctor；
- 析构函数：Finalize。

当字段初始化器与字段名称相同时，该名称将作为方法调用的一部分。

在两种情况下调用者成员信息不会生效。其一是特性初始化。代码清单16-3给出了一个特性示例，希望可以得到其应用到的成员名称，但遗憾的是编译器在这种情况下不会自动完成任何信息的填充。

代码清单16-3 试图在特性声明时使用调用者信息特性

```
[AttributeUsage(AttributeTargets.All)]
public class MemberDescriptionAttribute : Attribute
{
   public MemberDescriptionAttribute([CallerMemberName] string member = null)
   {
   Member = member;
   }

   public string Member { get; set; }
}
```

这本可以很有用。我曾多次见过开发者通过反射得到特性后，却不得不自己维护一个数据结构，以保存成员名和特性之间映射的例子，而这本可以由编译器自动完成。

特性对动态类型无效，这是可以原谅的。代码清单16-4展示了不能生效的情况。

代码清单16-4 试图在动态调用时使用调用者信息特性

```
class TypeUsedDynamically
{
   internal void ShowCaller([CallerMemberName] string caller = "Unknown")
   {
      Console.WriteLine("Called by: {0}", caller);
   }
}
...
dynamic x = new TypeUsedDynamically();
x.ShowCaller();
```

代码清单16-4只打印出了`Called by: Unknown`，仿若应用特性不存在一般。尽管看上去有点遗憾，但要想让它生效，编译器需在每个可能需要调用者信息的动态调用处都内嵌上成员名、

16

① 意思是指MemberName都返回成员名。——译者注

文件名和行数。总的来说，这对大多数开发者来说都是得不偿失的。

16.2.2　日志

调用者信息最明显的用途莫过于写入日志文件。以前记日志时，通常需要构造一个堆栈跟踪（如使用System.Diagnostics.StackTrace）来查找日志信息的出处。虽然它通常隐藏在日志框架的后台，但依然无法改变其丑陋的存在。此外，它还可能存在性能问题，并且在JIT编译器内联时十分脆弱。

不难想象日志框架会如何使用这个新特性，来低廉地记录调用者信息，即使某些程序集可能通过剥离调试信息或混淆操作来保护行数和成员名也无妨。当然，想记录完整的堆栈跟踪时，由于该特性起不到什么作用，因此需各位自行实现这一操作。

截至本书编写之时，还没有日志框架使用过该特性。首先它需要面向.NET 4.5进行构建，或者像16.2.4节介绍的那样，需要显式声明这些特性。不过为自己喜欢的日志框架编写一个包装类，并提供调用者信息还是很容易的。随着时间的推移，我敢肯定所有日志框架最终都会提供此种功能。

16.2.3　实现**INotifyPropertyChanged**

三大特性之一的[CallerMemberName]还有一个不太明显的用途，不过如恰好需要经常实现INotifyPropertyChanged的话，这种用法就显而易见了。

该接口十分简单，只包含一个类型为PropertyChangedEventHandler的事件。其委托类型签名如下：

```
public delegate void PropertyChangedEventHandler(Object sender,
                                                 PropertyChangedEventArgs e)
```

PropertyChangedEventArgs包含单一的构造函数：

```
public PropertyChangedEventArgs(string propertyName)
```

在C# 5之前，通常按以下方式实现INotifyPropertyChanged。

代码清单16-5　实现INotifyPropertyChanged的老办法

```
class OldPropertyNotifier : INotifyPropertyChanged
{
   public event PropertyChangedEventHandler PropertyChanged;

   private int firstValue;
   public int FirstValue
   {
      get { return firstValue; }
      set
      {
         if (value != firstValue)
         {
            firstValue = value;
```

```
                NotifyPropertyChanged("FirstValue");
            }
        }
    }

    // 其他属性也采用相同模式

    private void NotifyPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

辅助方法可避免在每个属性中都加入空验证。当然，也可以将其实现为扩展方法，以避免在每个实现类中都重复一遍。

这不仅冗长（此点没有改变），而且脆弱。问题在于属性的名称（`FirstValue`）指定为字符串字面量，而如果将属性名重构为其他名称，则很可能会忘记修改字符串字面量。幸运的话，工具和测试会帮助我们找到错误，但这仍然很丑陋。

在C# 5中，大部分代码仍然相同，但可在辅助方法中使用`CallerMemberName`，让编译器来填充属性名，如代码清单16-6所示。

代码清单16-6　使用调用者信息实现`INotifyPropertyChanged`

```
// 在setter里
if (value != firstValue)
{
    firstValue = value;
    NotifyPropertyChanged();
}

...

void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
{
    // 和之前一样的代码
}
```

此处只展示了发生变化的代码，就这么简单。现在如改变属性的名称，编译器则可用新名称进行替代。这并不是惊天动地的大改进，但却非常不错。

16.2.4　在非.NET 4.5 环境下使用调用者信息特性

与扩展方法一样，调用者信息特性也只是请求编译器在编译过程中进行代码的转换。该类特性并没有使用我们无法提供的信息，只是在使用时需格外小心。跟扩展方法一样，我们也可以在早期.NET版本中使用它们，只需自己声明这些特性即可，这就如同从MSDN中复制声明一样简单。这些特性本身不包含任何参数，所以在类声明中无须提供其他内容，但仍然要放在`System.`

Runtime.CompilerServices命名空间中。

C#编译器将按处理.NET 4.5中真正的调用者信息特性那样来处理用户提供的特性。这么做的缺点是，用.NET 4.5编译同样的代码时会产生错误。此时只需移除手动创建的特性，以避免编译器产生混淆即可。

如果使用的是.NET 4、Silverlight 4/5或Windows Phone 7.5，还可使用Microsoft.Bcl Nuget包。包内提供了这些特性，以及其他期待中的有用类型。

这就是有关C# 5的全部内容。

16.3　结束语

前两版《深入理解C#》都在结尾处用单独的一章，来展望当时我所能预见到的未来。如果你有其中一本（或两本都有），翻开看看，我敢肯定你会会心一笑。我不认为这当中存在着什么大的错误，但要我判断未来几年内会出现多大变化，这我可说不准。

我还要说明一点，在微软发布C# 4或C# 5之前，我并不知道它们会包含哪些内容。动态类型和异步函数对我来说是个莫大的惊喜。我曾非常荣幸地在一次会议上，当着一些C#小组成员的面，表达了我对C# 5未来的想法。我很庆幸他们没有按我说的思路走。我并没有明确地表达出将async/await作为一种特性，并且该特性远比我能想到的要高明得多。

信息产业会如何发展呢？更多的移动、触摸输入以及分布式云服务，可能还包括增强现实技术，这些都是十分安全的赌注[①]。但如果到2014年底，它们成为整个产业的颠覆性力量，我会感到非常失望。计算机最有魅力的部分似乎在于横空出世并震惊所有人，当然，这一切是建立在业内人士多年努力的基础之上的。

同样的事情也会发生在C#上。有一些微小特性我仍然希望可以在以后的版本中加以实现，也许C# 6就是这样的一个版本，即可包含许多微小特性，而不像此前版本一样只包含一些较大特性。或许C#将具有可扩展性，以允许其他开发者自行创建这些微小特性。又或许会诞生新的杀手级特性，我甚至可能都不知道我需要它。

C#和.NET团队当然没有闲着。即使抛开C# 5和与Windows 8 UI集成所需的全部工作，他们还在忙于一个项目，即Roslyn。其名称来自于Eric Lippert在处理该项目时办公室的朝向[②]；此外，Roslyn也是谈及很多年的“编译器即服务”（compiler as a service）思想的另一个名称。Roslyn将提供一个API，开发人员可用其分析C#（或VB）代码，以编程的方式进行修改，或将其编译为IL，等等。我觉得很少有开发者会需要这项功能，但那些需要的人必然会异常欣喜，并会为其他人创造出绝妙的东西来。试想一下，我们能够编写自己的重构工具、更复杂的代码约定分析工具、代码生成工具等，而所有这些都是基于一个API，它设计得非常强大和高性能，可以成为未来Visual Studio的引擎。也许对于大多数人来说更为重要的是，Roslyn为C#团队提供了一个摇篮，以便相对轻松地孕育出新的特性。也许它们在未来会更加大胆和雄心勃勃！

① 意思是指如果作者预测这些，肯定会在未来得到应验。——译者注

② 美国弗吉尼亚州罗斯林市。——译者注

可以肯定的是，不管C#是否还会继续发展，在相当长的时间内，我都将继续编写、谈论和使用C#。在未来十年内，编程都将会充满乐趣。

如同前两版中所讲到的，我建议你去做一些超赞的事情。编写清晰的代码，让你的同事愿意跟你合作；开发开源世界里的下一个奇迹；在Stack Overflow上帮助其他开发者；和用户、与会者、朋友，以及那些能够感受到你激情的人交谈。对于你的付出，我致以最美好的祝福。我也希望本书能够在你通往成功的道路上，对你有所助益。

附录 A

LINQ标准查询操作符

LINQ中有很多标准查询操作符，只有一些在C#查询表达式中被直接支持——其他一些必须作为普通方法来“手动”调用。某些标准查询操作符已经在本书的正文中演示过了，不过在这个附录中，还是被全部列出。我定义的两个示例序列如下：

```
string[] words = {"zero", "one", "two", "three", "four"};
int[] numbers = {0, 1, 2, 3, 4};
```

基于完整的目的，我包含了之前见过的一些操作符，不过大部分情况下，第11章包含的信息比这里提供的更详细。

此处指定的行为都是针对LINQ to Objects的，其他提供器可能不同。对于每个操作符，我都会说明，它是使用延迟执行还是立即执行的模式。如果一个操作符使用延迟执行，我同时会指出它使用的是流式数据还是缓冲数据。

刚才我在Edulinq项目中重新实现了LINQ to Objects，并将每一个操作符的细节内容都写到了博客里，同时考虑到了优化以及延迟计算等方面的可能性。可访问Edulinq项目主页http://edulinq.googlecode.com，了解更多LINQ to Objects的相关内容。

A.1　聚合

聚合操作符（见表A-1），所有的结果只有一个值而不是一个序列。Average和Sum针对数值（任何内置数值类型）序列或使用委托从元素值转换为内置数值类型的元素序列。Min和Max具有不同数值类型的重载，不过也只能在对元素类型使用默认比较符或使用转换委托的序列上进行操作。Count和LongCount是等价的，不同之处仅仅在于返回类型。它们两者都具有两个重载——一个只统计序列长度，一个可以接受谓词，即只统计与谓词匹配的元素。

表A-1　聚合操作符示例

表　达　式	结　　果
numbers.Sum()	10
numbers.Count()	5
numbers.Average()	2
numbers.LongCount(x => x % 2 == 0)	3（long类型；有3个偶数）

（续）

表　达　式	结　　果
`words.Min(word => word.Length)`	3（"one"和"two"）
`words.Max(word => word.Length)`	5（"three"）
`numbers.Aggregate("seed", (current,item) => current + item, result => result.ToUpper())`	"SEED01234"

最常见的聚合操作符就是Aggregate（表A-1最后一行）。所有的其他聚合操作符都能表示为对Aggregate的调用，虽然这样做会相对繁琐。其基本思想就是，总是存在以初始元素开头的"当前结果"。聚合委托被应用于输入序列的每个元素；委托取得当前结果和输入元素，并生成下一个结果。作为最终可选步骤，转换被应用于聚合结果上，并将其转换为这个方法的返回值。如果有必要，这种转换可以产生不同的数据类型。这虽然不像听起来那么复杂，不过你依旧不希望频繁地使用它。

所有的聚合操作符都使用立即执行的模式。没有使用谓词的Count方法的重载为实现了ICollection和ICollection<T>的类型进行了优化。届时，将使用集合的Count属性，无须读取任何数据[①]。

A.2　连接

只存在一个连接操作符：Concat（见表A-2）。如你所料，它会对两个序列进行操作，并返回一个单独的序列，该序列中包含了第1个序列和第2个序列中所有元素，且第2个序列连接在第1个序列的后面。两个输入序列必须具有相同的类型，使用延迟执行模式，并且均为流式数据。

表A-2　**Concat**示例

表　达　式	结　　果
`numbers.Concat(new[] {2, 3, 4, 5, 6})`	0, 1, 2, 3, 4, 2, 3, 4, 5, 6

A.3　转换

转换操作符被广泛地使用，不过它们总是成对出现。表A-3使用了另外两个序列来解释Cast和OfType。

```
object[] allStrings = {"These", "are", "all", "strings"};
object[] notAllStrings = {"Number", "at", "the", "end", 5};
```

① LongCount没有这种优化。我个人从来没有在LINQ to Objects中使用过这个方法。

表A-3　转换示例

表　达　式	结　　果
allStrings.Cast<string>()	"These", "are", "all", "strings"（作为IEnumerable<string>）
allStrings.OfType<string>()	"These", "are", "all", "strings"（作为IEnumerable<string>）
notAllStrings.Cast<string>()	遍历时遇到转换失败的地方，会抛出异常
notAllStrings.OfType<string>()	"Number", "at", "the", "end"（作为IEnumerable<string>）
numbers.ToArray()	0, 1, 2, 3, 4（作为int[]）
numbers.ToList()	0, 1, 2, 3, 4（作为List<int>）
words.ToDictionary (w =>w.Substring(0, 2))	Dictionary中的内容： "ze": "zero" "on": "one" "tw": "two" "th": "three" "fo": "four"

// 键是这个单词的第一个字母
words.ToLookup(word => word[0])
	Lookup中的内容：
	'z': "zero"
	'o': "one"
	't': "two", "three"
	'f': "four"

words.ToDictionary(word => word[0])	异常：每个键只能有一个数据项，所以在遇到't'时转换失败

ToArray和ToList的含义显而易见：它们读取整个序列到内存中，并把结果作为一个数组或一个List<T>返回。两者都是立即执行。

Cast和OfType把一个非类型化序列转换为类型化的，或抛出异常（对于Cast），或忽略那些不能隐式转换为输出序列元素类型的输入序列元素（对于OfType）。这个运算符也能用于把某个类型化序列转换为更加具体的类型化序列，例如把IEnumerable<object>转换为IEnumerable<string>。转换以流的方式延迟执行。

ToDictionary和ToLookup都使用委托来获得任何特定元素的键。ToDictionary返回一个把键映射到元素类型的字典，而ToLookup返回相应的类型化的ILookup<,>。查找类似于查字典，只不过和健相关的值不是一个元素而是元素的序列。查找通常用于普通操作中希望有重复的键存在的时候，而重复的键会引起ToDictionary抛出异常。两者更复杂的重载方法可以将自定义的IEqualityComparer<T>用于比较键的操作，并在每个元素被放到字典或者开始查找之前，把转换委托应用于其上。另外，两个方法都会使用立即执行的模式。

我没有提供AsEnumerable和AsQueryable这两个操作符的例子，因为它们不会以一种显而易见的方式立即影响结果。但它们影响查询执行的方式。Queryable.AsQueryable是返回一个IQueryable类型（既可以是泛型，也可以是非泛型，取决于你选取的重载）的IEnumerable上的扩展方法。如果调用的IEnumerable已经是一个IQueryable，则它会返回同一个引用，否则就创建一个包装类来包含原始序列。通过这个包装类，你可以使用所有普通的Queryable扩展方法，在表达式树中传递，不过在查询执行的时候，表达式树被编译为普通的IL代码直接执行。LambdaExpression.Compile的用法已经在9.3.2节中讲到过。

Enumerable.AsEnumerable是IEnumerable<T>的扩展方法，包含一些简单的实现，仅返回被调用者的引用。这次不会用到包装类——仅返回同一个引用。它强制Enumerable扩展方法用于随后的LINQ操作符。思考一下如下查询表达式：

```
// Filter the users in the database with LIKE
from user in context.Users
where user.Name.StartsWith("Tim")
select user;

// Filter the users in memory
from user in context.Users.AsEnumerable()
where user.Name.StartsWith("Tim")
select user;
```

第2个查询表达式强制编译时的源类型为IEnumerable<User>而非IQueryable <User>，因而所有的处理过程都可以在内存中而不必在数据库中完成。编译器将使用Enumerable的扩展方法（它获取委托参数）而不是Queryable的扩展方法（它获取表达式树参数）。通常而言，你希望尽可能在SQL中完成大多数处理，但在遇到需要“本地”代码参与转换的时候，你有时不得不强制LINQ去使用适当的Enumerable扩展方法。当然，这不仅仅针对数据库，其他提供器也可以在查询末端强制使用Enumerable，只要它们基于IQueryable及其同类即可。

A.4　元素操作符

另外一种用于选择数据的查询操作符，分组后成对显示在表A-4中。这次，每一对操作符都以同样的方式执行。选取单个元素的简化版本，就是在特定元素存在时返回它，不存在时就抛出一个异常，还有一个以OrDefault结尾的版本。OrDefault版本的功能完全一样，只是在找不到想要的元素时，返回结果类型的默认值，而非抛出一个异常。所有操作符都使用立即执行的模式。

表A-4　单个元素选取示例

表　达　式	结　　果
words.ElementAt(2)	"two"
words.ElementAtOrDefault(10)	null
words.First()	"zero"
words.First(w => w.Length == 3)	"one"
words.First(w => w.Length == 10)	异常：没有匹配的元素
words.FirstOrDefault (w => w.Length == 10)	null
words.Last()	"four"
words.Single()	异常：不止一个元素
words.SingleOrDefault()	异常：不止一个元素
words.Single(word => word.Length == 5)	"three"
words.Single(word => word.Length == 10)	异常：没有匹配的元素
words.SingleOrDefault(w => w.Length == 10)	null

操作符的名称很容易理解：First和Last分别返回序列的第1个和最后1个元素，如果序列为空，抛出InvalidOperationException异常。Single返回序列中的一个元素，如果序列为空或者返回一个以上的元素则抛出异常，而ElementAt通过索引返回特定的元素（例如，第5个元素）。如果索引为负数或大于集合中元素的实际数量，将抛出ArgumentOutOfRangeException。另外，除了ElementAt首先过滤序列以外，所有的操作符都存在相应的重载方法——例如，First可以返回匹配给定条件的第1个元素。

以上这些方法的OrDefault版本都不会抛出异常，而是返回元素类型的默认值。但有一种例外情况：如果序列为空（empty），SingleOrDefault将返回默认值，但如果序列中的元素不止一个，将抛出异常，就像Single方法一样。该方法适用于所有条件正确，序列只包含0或1个元素的情况。如果你要处理的序列可能包含多个元素，应该使用FirstOrDefault方法。

所有没有使用谓词参数的重载都为IList<T>的实例进行了优化，因为它们不通过迭代就可以访问正确的元素。包含谓词的版本没有优化，因为这对大多数调用都没有意义，尽管从末尾向后移动，会使查找列表中最后一个匹配元素的过程产生很大不同。在本书写作之时，这种情况还没有优化，未来版本也许会发生改变。

A.5　相等操作

只有一个标准的相等操作符：SequenceEqual（见表A-5）。它是按照顺序逐一比较两个序列中的元素是否相等。例如，序列0，1，2，3，4不等于4，3，2，1，0。重载方法允许使用具体的IEqualityComparer<T>，对元素进行比较。返回值就是一个Boolean值，并使用立即执行的模式来计算。

表A-5　序列相等运算示例

表　达　式	结　　果
`words.SequenceEqual (new[]{"zero","one", "two","three","four"})`	True
`words.SequenceEqual (new[]{"ZERO","ONE","TWO","THREE","FOUR"})`	False
`words.SequenceEqual (new[]{"ZERO","ONE", "TWO","THREE","FOUR"}, StringComparer.OrdinalIgnoreCase)`	True

同样，LINQ to Objects在这里也没有进行优化：如果存在有效的方式可以得到两个序列的数量，那么在比较它们的元素之前，就能知道它们是否相等了。而实际上，该方法的实现直接遍历两个序列，直到末尾或找到不等的元素。

A.6　生成

在所有的生成操作符（见表A-6）中，只有一个会对现有的序列进行处理：DefaultIfEmpty。

如果序列不为空，就返回原始序列，否则返回含有单个元素的序列。其中的元素通常是序列类型的默认值，不过重载方法允许你设定要使用的值。

表A-6　生成操作符示例

表　达　式	结　　果
`numbers.DefaultIfEmpty()`	`0, 1, 2, 3, 4`
`new int[0].DefaultIfEmpty()`	`0`（包含在一个`IEnumerable<int>`类型的序列中）
`new int[0].DefaultIfEmpty(10)`	`10`（包含在一个`IEnumerable<int>`类型的序列中）
`Enumerable.Range(15, 2)`	`15, 16`
`Enumerable.Repeat(25, 2)`	`25, 25`
`Enumerable.Empty<int>()`	一个类型为`IEnumerable<int>`的空序列

其他3个生成操作符仅仅是`Enumerable`的静态方法。

- `Range`生成一个整数序列，通过参数可以设定第一个值和需要生成值的个数。
- `Repeat`根据指定的次数来重复特定的单个值，以生成任意类型的序列。
- `Empty`生成任意类型的空序列。

所有生成操作符都使用延迟执行，并对结果进行流式处理。也就是说，它们不会预先生成集合并返回。不过，返回正确类型的空数组的`Empty`方法是个例外。一个空的数组是完全不可变的，因此相同元素类型的所有这种调用，都将返回相同的空数组。

A.7　分组

有两个分组操作符，不过其中一个就是`ToLookup`（我们已经在A-3中讲述转换操作符的时候看到过了）。剩下的`GroupBy`，已经在11.6.1节中讨论查询表达式中的`group... by`子句时见过。它使用延迟执行，不过会缓存结果。当你开始迭代分组的结果序列时，消费的是整个输入序列。

`GroupBy`的结果是相应类型化`IGrouping<,>`元素的序列。每个元素具有一个键和与之匹配的元素序列。从许多方面讲，它只是查找的一种不同的方式——不是通过键随机访问分组数据，而是顺序枚举它。分组数据的返回顺序是它们各自键被发现的顺序。在一个分组数据内，元素的顺序和最初序列中的一致。

`GroupBy`具有大量的重载方法，你既可以设定从元素派生出键的方式（这一项是必须指定的），也可以有选择地设定如下内容。

- 如何比较键。
- 从原始元素到分组内元素的投影。
- 包含键和匹配元素序列的投影。这时，整个结果为投影结果类型的元素序列。

表A-7包含了第二个和第三个选项的示例，当然还有最简单的形式。自定义键比较器不是三言两语就能概括的，不过它们的工作方式一目了然。

表A-7　**GroupBy**示例

表　达　式	结　　果

`words.GroupBy(word => word.Length)`	`Key: 4; Sequence: "zero", "four"`
	`Key: 3; Sequence: "one", "two"`
	`Key: 5; Sequence: "three"`

`words.GroupBy`
`  (word => word.Length, // 键`
`  word => word.ToUpper() // 分组元素`
` )`
	`Key: 4; Sequence: "ZERO", "FOUR"`
	`Key: 3; Sequence: "ONE", "TWO"`
	`Key: 5; Sequence: "THREE"`

`// Project each (key, group) pair to string`
`words.GroupBy`
`  (word => word.Length,`
`   (key, g) => key + ": " + g.Count())`
	`"4: 2", "3: 2", "5: 1"`

在我的印象中，最后一个选项很少使用。

A.8　连接

有两个操作符用于连接，即`Join`和`GroupJoin`，我们在11.5节分别使用`join`和`join...into`查询表达式子句的时候看到过这两个操作符。每个方法都可以获取几个参数：两个序列、用于每个序列的键选择器，应用于每个匹配元素对的投影，以及可选的键比较器。

对于`Join`，投影从每个序列中获取一个元素，并生成一个结果；对于`GroupJoin`，投影从左边序列获取一个元素，然后从右边序列获取一个由匹配元素构成的序列。两者都是延迟执行，并流处理左边序列，而对于右边序列，在请求第一个结果时便读取其全部内容。

对于在表A-8的连接例子中，我们将按照姓名和颜色的第一个字母把一个姓名序列（Robin、Ruth、Bob、Emma）匹配到一个颜色序列（Red、Blue、Beige、Green），例如，Robin将会和Red连接上，Blue和Beige连接。

表A-8　**Join**示例

表　达　式	结　　果

`names.Join // 左边序列`
`  (colors, // 右边序列`
`  name => name[0], // 左边键选择器`
`  color => color[0], // 右边键选择器`
`  // 为结果对投影`
`  (name, color) => name+" - " + color`
` )`
	`"Robin - Red",`
	`"Ruth - Red",`
	`"Bob - Blue",`
	`"Bob - Beige"`

`Names.GroupJoin`
`  (colors,`
`  name => name[0],`
`  color => color[0],`
`  // 为键/序列对投影`
`  (name, matches) => name + ": " +`
`     string.Join("/", matches.ToArray())`
` )`
	`"Robin: Red",`
	`"Ruth: Red",`
	`"Bob: Blue/Beige",`
	`"Emma: "`

注意，Emma未匹配任何颜色——所以，该姓名不会显示在第一个例子的结果中，但却会显示在第二个例子中，只是颜色序列是空的。

A.9　分部

分部操作符中，既可以跳过（skip）序列的开始部分，只返回剩余元素，也可以只获取（take）序列的开始部分，而忽略剩余元素。在每种情况下，你都可以设置序列的第一部分包含多少个元素，或设定相应的条件——只要条件满足，序列的第一部分就继续往前进行。在条件第一次不满足时，就不再往下判断——而不管序列的后面元素是否匹配。所有的部分操作符（见表A-9）都是延迟执行和流式数据。

通过位置或谓词，分组可以将一个序列有效地划分成两个单独的部分。在这两种情况下，如果你将`Take`或`TakeWhile`的结果与相对应的`Skip`或`SkipWhile`的结果进行连接，并为两种调用提供相同的参数，将得到原始序列：每个元素都按原始顺序出现一次。表A-9演示了这种调用。

表A-9　分部示例

表　达　式	结　　果
`words.Take(2)`	`"zero", "one"`
`words.Skip(2)`	`"two","three", "four"`
`words.TakeWhile(word => word.Length <= 4)`	`"zero", "one", "two"`
`words.SkipWhile(word => word.Length <= 4)`	`"three","four"`

A.10　投影

我们在第11章看到过这两个投影操作符（`Select`和`SelectMany`）。`Select`是一种简单的从源元素到结果元素的一对一投影。`SelectMany`在查询表达式中有多个`from`子句的时候使用；原始序列中的每个元素都用来生成新的序列。两个投影操作符（见表A-10）都是延迟执行。

表A-10　投影示例

表　达　式	结　　果
`words.Select(word => word.Length)`	`4, 3, 3, 5, 4`
`words.Select ((word, index) => index.ToString() + ": " +word)`	`"0: zero", "1: one", "2: two", "3: three", "4: four"`
`words.SelectMany (word => word.ToCharArray())`	`'z', 'e', 'r', 'o', 'o', 'n', 'e', 't', 'w', 'o', 't', 'h', 'r', 'e', 'e', 'f', 'o', 'u', 'r'`
`words.SelectMany ((word, index) => Enumerable.Repeat(word, index))`	`"one", "two", "two", "three", "three","three", "four", "four","four","four"`

这里包含了第11章没有介绍的重载。这两种方法都含有允许在投影中使用原始序列索引的重

载，并且SelectMany还可以将所有生成的序列合并成一个单独的序列，而不包含原始序列[①]，或使用投影来为每个元素对生成结果元素[②]。多个from子句通常使用的是包含投影的重载。（篇幅所限，恕不举例，详细内容参见第11章。）

.NET 4引入了一个新的操作符Zip。根据MSDN所述，它并不是一个标准的查询操作符，但还是有必要了解一下。它包含两个序列，并对每个元素对应用指定的投影：先是每个序列的第一个元素，然后是每个序列的第二个元素，以此类推。任何一个源序列达到末尾时，结果序列都将停止产生。表A-11展示了Zip的两个示例，使用了A.8节中的名称和颜色。Zip使用了延迟执行和流式数据。

表A-11　ZiP示例

表　达　式	结　　果
`names.Zip(colors, (x, y) => x+ "-" +y)`	"Robin-Red", "Ruth-Blue", "Bob-Beige", "Emma-Green"

`// 第二个序列提前停止生成`
`names.Zip(colors.Take(3),`
`  (x, y) => x + "-" + y`
	"Robin-Red",
	"Ruth-Blue",
	"Bob-Beige"

A.11　数量

数量操作符（见表A-12）都返回一个Boolean值，使用立即执行。

- All操作符检查在序列中的所有元素是否满足指定的条件。
- Any操作符检查在序列中的任意元素是否满足指定的谓词，如果使用没有参数的重载，则检查序列中是否存在元素。
- Contains操作符检查序列是否包含特殊的元素，可选设置要使用的比较方式。

表A-12　数量示例

表　达　式	结　　果
`words.All(word => word.Length > 3)`	false（"one"和"two"的确包含3个字母）
`words.All(word => word.Length > 2)`	true
`words.Any()`	true（序列不为空）
`words.Any(word => word.Length == 6)`	false（没有6个字母的单词）
`words.Any(word => word.Length == 5)`	true（"three"满足这个条件）
`words.Contains("FOUR")`	false
`words.Contains("FOUR",` `  StringComparer.OrdinalIgnoreCase) )`	true

① 如表A-10中第一个SelectMany的示例。——译者注
② 如表A-10中第二个SelectMany的示例。——译者注

A.12　过滤

两个过滤操作符是OfType和Where。有关OfType操作符的细节和例子，请参见转换操作符一节（A.3节）。Where运算符返回一个序列，这个序列包括与给定谓词匹配的所有元素。它还有一个重载方法，以便谓词能使用元素索引。通常是不需要索引的，而在查询表达式中的where子句也不使用这个重载方法。Where总是使用延迟执行和流式数据，表A-13列出了两个重载方法。

表A-13　过滤示例

表　达　式	结　　果
words.Where(word => word.Length > 3)	"zero", "three", "four"
words.Where ((word, index) => index < word.Length)	"zero", // index=0, length=4 "one", // index=1, length=3 "two", // index=2, length=3 "three", // index=3, length=5 // Not "four", index=4, length=4

A.13　基于集的操作符

把两个序列看作元素的集合是很自然的。4个基于集合的运算符都具有两个重载方法，一个使用元素类型的默认相等比较，一个用于在额外的参数中指定比较。所有的集合运算符都是延迟执行。

Distinct操作符最简单——它只对单个序列起作用，并且返回所有不重复元素（已经排除了重复项）的新序列。其他的运算符也确保只返回不重复的元素，不过它们对两个序列起作用。

- Intersect返回在两个序列中都出现的元素。
- Union返回出现在任一序列中的元素。
- Except返回出现在第一个序列，但不出现在第二个序列中的元素（出现第二个序列但不在第一个中的元素也不返回）。

表A-14中这些操作符的示例使用了两个新序列，即abbc（"a","b","b","c"）和cd（"c","d"）。

表A-14　基于集合的操作符示例

表　达　式	结　　果
abbc.Distinct()	"a", "b", "c"
abbc.Intersect(cd)	"c"
abbc.Union(cd)	"a", "b", "c", "d"
abbc.Except(cd)	"a", "b"
cd.Except(abbc)	"d"

所有这些操作符都使用延迟执行，但有的使用了缓冲，有的使用了流式处理，缓冲和流式处理显然更复杂一些。Distinct和Union都对输入序列进行了流式处理，而Intersect和Except

先是读取整个右边输入序列，然后像连接操作符那样对左边输入序列进行流式处理[1]。所有这些操作符都保存已返回元素的集，以确保不返回重复的元素。这意味着即使是Distinct和Union，也不适合那些过大而不适于放入内存的序列，除非你知道经过处理后的集并不会包含很多元素。

A.14　排序

我们之前见过所有的排序运算符：OrderBy和OrderByDescending提供了“主要的”排序方式，而ThenBy和ThenByDescending提供了次要的排序方式，用以区别使用主要的排序方式无法区别的元素。在每种情况中，都要指定从元素到排序键的投影，也指定键之间的比较。不像框架中的其他排序算法（比如List<T>.Sort），LINQ排序比较稳定——换句话说，如果两个元素根据它们的排序关键字被认为是相等的，那么将按照它们在原始序列中的顺序返回。

最后一个排序操作符是Reverse，它仅反转序列的顺序。所有的排序操作符（见表A-15）都是延迟执行，不过会缓存其中的数据。

表A-15　排序示例

表　达　式	结　　果
`words.OrderBy(word => word)`	"four", "one","three", "two", "zero"
`// 依据第二个字母对单词排序` `words.OrderBy(word => word[1])`	"zero", "three","one", "four", "two"
`// 依据长度对单词排序，如长度相同则以原顺序返回` `words.OrderBy(word => word.Length)`	"one", "two","zero", "four", "three"
`words.OrderByDescending (word => word.Length)`	"three", "zero", s "four", "one", "two"
`// 依据长度排序，如长度相同则以首字母顺序排序` `words.OrderBy(word => word.Length) .ThenBy(word => word)`	"one", "two","four", "zero", "three"
`// 依据长度排序，如长度相同则按反向字母顺序排序` `words.OrderBy(word => word.Length) .ThenByDescending(word => word)`	"two", "one","zero", "four", "three"
`words.Reverse()`	"four", "three", "two", "one", "zero"

① 对于abbc.Intersect(cd)来说，cd为右边输入序列，abbc为左边输入序列。——译者注

附录 B

.NET中的泛型集合

.NET中包含很多泛型集合，并且随着时间的推移列表还在增长。本附录涵盖了最重要的泛型集合接口和类，但不会涉及System.Collections、System.Collections.Specialized和System.ComponentModel中的非泛型集合。同样，也不会涉及ILookup<TKey,TValue>这样的LINQ接口。本附录是参考而非指南——在写代码时，可以用它来替代MSDN。在大多数情况下，MSDN显然会提供更详细的内容，但这里的目的是在选择代码中要用的特定集合时，可以快速浏览不同的接口和可用的实现。

我没有指出各集合是否为线程安全，MSDN中有更详细的信息。普通的集合都不支持多重并发写操作；有些支持单线程写和并发读操作。B.6节列出了.NET 4中添加的并发集合。此外，B.7节介绍了.NET4.5中引入的只读集合接口。

B.1 接口

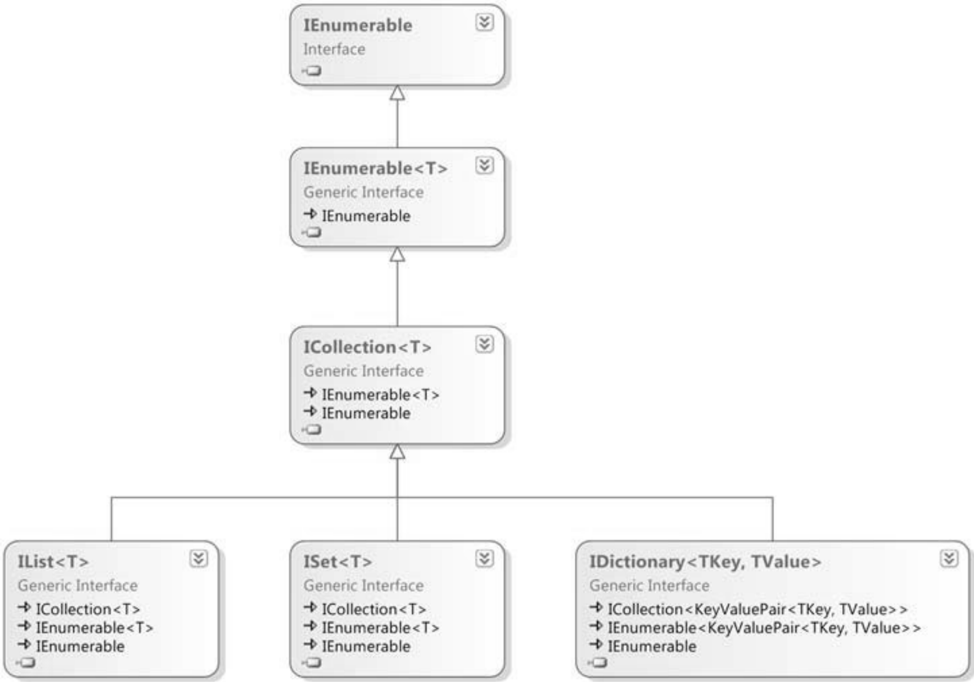
几乎所有要学习的接口都位于System.Collections.Generic命名空间。图B-1展示了.NET4.5以前主要接口间的关系，此外还将非泛型的IEnumerable作为根接口包括了进来。为避免图表过于复杂，此处没有包含.NET 4.5的只读接口。

正如我们已经多次看到的，最基础的泛型集合接口为IEnumerable<T>，表示可迭代的项的序列。IEnumerable<T>可以请求一个IEnumerator<T>类型的迭代器。由于分离了可迭代序列和迭代器，这样多个迭代器可以同时独立地操作同一个序列。如果从数据库角度来考虑，表就是IEnumerable<T>，而游标是IEnumerator<T>。本附录仅有的两个可变（variant）集合接口为.NET 4中的IEnumerable<out T>和IEnumerator<out T>；其他所有接口的元素类型值均可双向进出，因此必须保持不变。

接下来是ICollection<T>，它扩展了IEnumerable<T>，添加了两个属性（Count和IsReadOnly）、变动方法（Add、Remove和Clear）、CopyTo（将内容复制到数组中）和Contains（判断集合是否包含特殊的元素）。所有标准的泛型集合实现都实现了该接口。

IList<T>全都是关于定位的：它提供了一个索引器、InsertAt和RemoveAt（分别与Add和Remove相同，但可以指定位置），以及IndexOf（判断集合中某元素的位置）。对IList<T>进行迭代时，返回项的索引通常为0、1，以此类推。文档里没有完整的记录，但这是个合理的假

设。同样，通常认为可以快速通过索引对IList<T>进行随机访问。



图B-1　System.Collections.Generic中的接口（不包括.NET 4.5）

IDictionary<TKey, TValue>表示一个独一无二的键到它所对应的值的映射。值不必是唯一的，而且也可以为空；而键不能为空。可以将字典看成是键/值对的集合，因此IDictionary<TKey, TValue>扩展了ICollection<KeyValuePair<TKey, TValue>>。获取值可以通过索引器或TryGetValue方法；与非泛型IDictionary类型不同，如果试图用不存在的键获取值，IDictionary<TKey, TValue>的索引器将抛出一个KeyNotFoundException。TryGetValue的目的就是保证在用不存在的键进行探测时还能正常运行。

ISet<T>是.NET 4新引入的接口，表示唯一值集。它反过来应用到了.NET 3.5中的HashSet<T>上，以及.NET 4引入的一个新的实现——SortedSet<T>。

在实现功能时，使用哪个接口（甚至实现）是十分明显的。难的是如何将集合作为API的一部分公开；返回的类型越具体，调用者就越依赖于你指定类型的附加功能。这可以使调用者更轻松，但代价是降低了实现的灵活性。我通常倾向于将接口作为方法和属性的返回类型，而不是保证一个特定的实现类。在API中公开易变集合之前，你也应该深思熟虑，特别是当集合代表的是对象或类型的状态时。通常来说，返回集合的副本或只读的包装器是比较适宜的，除非方法的全部目的就是通过返回集合做出变动。

B.2　列表

从很多方面来说，列表是最简单也最自然的集合类型。框架中包含很多实现，具有各种功能和性能特征。一些常用的实现在哪里都可以使用，而一些较有难度的实现则有其专门的使用场景。

B.2.1　List<T>

在大多数情况下，List<T>都是列表的默认选择。它实现了IList<T>，因此也实现了ICollection<T>、IEnumerable<T>和IEnumerable。此外，它还实现了非泛型的ICollection和IList接口，并在必要时进行装箱和拆箱，以及进行执行时类型检查，以保证新元素始终与T兼容。

List<T>在内部保存了一个数组，它跟踪列表的逻辑大小和后台数组的大小。向列表中添加元素，在简单情况下是设置数组的下一个值，或（如果数组已经满了）将现有内容复制到新的更大的数组中，然后再设置值。这意味着该操作的复杂度为O(1)或O(n)，取决于是否需要复制值。扩展策略没有在文档中指出，因此也不能保证——但在实践中，该方法通常可以扩充为所需大小的两倍。这使得向列表末尾附加项为O(1)平摊复杂度（amortized complexity）；有时耗时更多，但这种情况会随着列表的增加而越来越少。

你可以通过获取和设置Capacity属性来显式管理后台数组的大小。TrimExcess方法可以使容量等于当前的大小。实战中很少有必要这么做，但如果在创建时已经知道列表的实际大小，则可将初始的容量传递给构造函数，从而避免不必要的复制。

从List<T>中移除元素需要复制所有的后续元素，因此其复杂度为O($n-k$)，其中k为移除元素的索引。从列表尾部移除要比从头部移除廉价得多。另一方面，如果要通过值移除元素而不是索引（通过Remove而不是RemoveAt），那么不管元素位置如何复杂度都为O(n)：每个元素都将得到平等的检查或打乱。

List<T>中的各种方法在一定程度上扮演着LINQ前身的角色。ConvertAll可进行列表投影；FindAll对原始列表进行过滤，生成只包含匹配指定谓词的值的新列表。Sort使用类型默认的或作为参数指定的相等比较器进行排序。但Sort与LINQ中的OrderBy有个显著的不同：Sort修改原始列表的内容，而不是生成一个排好序的副本。并且，Sort是不稳定的，而OrderBy是稳定的；使用Sort时，原始列表中相等元素的顺序可能会不同。LINQ不支持对List<T>进行二进制搜索：如果列表已经按值正确排序了，BinarySearch方法将比线性的IndexOf搜索效率更高[①]。

List<T>中略有争议的部分是ForEach方法。顾名思义，它遍历一个列表，并对每个值都执行某个委托（指定为方法的参数）。很多开发者要求将其作为IEnumerable<T>的扩展方法，但却一直没能如愿；Eric Lippert在其博客中讲述了这样做会导致哲学麻烦的原因（参见http://mng.bz/Rur2）。在我看来使用Lambda表达式调用ForEach有些矫枉过正。另一方面，如果你已经拥有一个要为列表中每个元素都执行一遍的委托，那还不如使用ForEach，因为它已经存在了。

① 二进制搜索的复杂度为O(log n)，线性搜索为O(n)。

B.2.2　数组

在某种程度上，数组是.NET中最低级的集合。所有数组都直接派生自`System.Array`，也是唯一的CLR直接支持的集合。一维数组实现了`IList<T>`（及其扩展的接口）和非泛型的`IList`、`ICollection`接口；矩形数组只支持非泛型接口。数组从元素角度来说是易变的，从大小角度来说是固定的。它们显示实现了集合接口中所有的可变方法（如`Add`和`Remove`），并抛出`NotSupportedException`。

引用类型的数组通常是协变的；如`Stream[]`引用可以隐式转换为`Object[]`，并且存在显式的反向转换[①]。这意味着将在执行时验证数组的改变——数组本身知道是什么类型，因此如果先将`Stream[]`数组转换为`Object[]`，然后再试图向其存储一个非`Stream`的引用，则将抛出`ArrayTypeMismatchException`。

CLR包含两种不同风格的数组。向量是下限为0的一维数组，其余的统称为数组（array）。向量的性能更佳，是C#中最常用的。`T[][]`形式的数组仍然为向量，只不过元素类型为`T[]`；只有C#中的矩形数组，如`string[10, 20]`，属于CLR术语中的数组。在C#中，你不能直接创建非零下限的数组——需要使用`Array.CreateInstance`来创建，它可以分别指定下限、长度和元素类型。如果创建了非零下限的一维数组，就无法将其成功转换为`T[]`——这种强制转换可以通过编译，但会在执行时失败。

C#编译器在很多方面都内嵌了对数组的支持。它不仅知道如何创建数组及其索引，还可以在`foreach`循环中直接支持它们；在使用表达式对编译时已知为数组的类型进行迭代时，将使用`Length`属性和数组索引器，而不会创建迭代器对象。这更高效，但性能上的区别通常忽略不计。

与`List<T>`相同，数组支持`ConvertAll`、`FindAll`和`BinarySearch`方法，不过对数组来说，这些都是`Array`类的以数组为第一个参数的静态方法。

回到本节最开始所说的，数组是相当低级的数据结构。它们是其他集合的重要根基，在适当的情况下有效，但在大量使用之前还是应该三思。Eric同样为该话题撰写了博客，指出它们有“些许害处”（参见http://mng.bz/3jd5）。我不想夸大这一点，但在选择数组作为集合类型时，这是一个值得注意的缺点。

B.2.3　`LinkedList<T>`

什么时候列表不是list呢？答案是当它为链表的时候。`LinkedList<T>`在很多方面都是一个列表，特别的，它是一个保持项添加顺序的集合——但它却没有实现`IList<T>`。因为它无法遵从通过索引进行访问的隐式契约。它是经典的计算机科学中的双向链表：包含头节点和尾节点，每个节点都包含对链表中前一个节点和后一个节点的引用。每个节点都公开为一个`LinkedListNode<T>`，这样就可以很方便地在链表的中部插入或移除节点。链表显式地维护其大小，因此可以访问`Count`属性。

① 容易混淆的是，也可以将`Stream[]`隐式转换为`IList<Object>`，尽管`IList<T>`本身是不变的。

在空间方面，链表比维护后台数组的列表效率要低，同时它还不支持索引操作，但在链表中的任意位置插入或移除元素则非常快，前提是只要在相关位置存在对该节点的引用。这些操作的复杂度为O(1)，因为所需要的只是对周围的节点修改前/后的引用。插入或移除头尾节点属于特殊情况，通常可以快速访问需要修改的节点。迭代（向前或向后）也是有效的，只需要按引用链的顺序即可。

尽管LinkedList<T>实现了Add等标准方法（向链表末尾添加节点），我还是建议使用显式的AddFirst和AddLast方法，这样可以使意图更清晰。它还包含匹配的RemoveFirst和RemoveLast方法，以及First和Last属性。所有这些操作返回的都是链表中的节点而不是节点的值；如果链表是空（empty）的，这些属性将返回空（null）。

B.2.4　Collection<T>、BindingList<T>、ObservableCollection<T>和KeyedCollection<TKey, TItem>

Collection<T>与我们将要介绍的剩余列表一样，位于System.Collections.ObjectModel命名空间。与List<T>类似，它也实现了泛型和非泛型的集合接口。

尽管你可以对其自身使用Collection<T>，但它更常见的用法是作为基类使用。它常扮演其他列表的包装器的角色：要么在构造函数中指定一个列表，要么在后台新建一个List<T>。所有对于集合的变动行为，都通过受保护的虚方法（InsertItem、SetItem、RemoveItem和ClearItems）实现。派生类可以拦截这些方法，引发事件或提供其他自定义行为。派生类可通过Items属性访问被包装的列表。如果该列表为只读，公共的变动方法将抛出异常，而不再调用虚方法，你不必在覆盖的时候再次检查。

BindingList<T>和ObservableCollection<T>派生自Collection<T>，可以提供绑定功能。BindingList<T>在.NET 2.0中就存在了，而ObservableCollection<T>是WPF（Windows Presentation Foundation）引入的。当然，在用户界面绑定数据时没有必要一定使用它们——你也许有自己的理由，对列表的变化更有兴趣。这时，你应该观察哪个集合以更有用的方式提供了通知，然后再选择使用哪个。注意，只会通知你通过包装器所发生的变化；如果基础列表被其他可能会修改它的代码共享，包装器将不会引发任何事件。

KeyedCollection<TKey, TItem>是列表和字典的混合产物，可以通过键或索引来获取项。与普通字典不同的是，键不能独立存在，应该有效地内嵌在项中。在许多情况下，这很自然，例如一个拥有CustomerID属性的Customer类型。KeyedCollection<,>为抽象类；派生类将实现GetKeyForItem方法，可以从列表中的任意项中提取键。在我们这个客户的示例中，GetKeyForItem方法返回给定客户的ID。与字典类似，键在集合中必须是唯一的——试图添加具有相同键的另一个项将失败并抛出异常。尽管不允许空键，但GetKeyForItem可以返回空（如果键类型为引用类型），这时将忽略键（并且无法通过键获取项）。

B.2.5　ReadOnlyCollection<T>和ReadOnlyObservableCollection<T>

最后两个列表更像是包装器，即使基础列表为易变的也只提供只读访问。它们仍然实现了泛

型和非泛型的集合接口。并且混合使用了显式和隐式的接口实现，这样使用具体类型的编译时表达式的调用者将无法使用变动操作。

`ReadOnlyObservableCollection<T>`派生自`ReadOnlyCollection<T>`，并和`ObserverbleCollection<T>`一样实现了相同的`INotifyCollectionChanged`和`INotifyPro pertyChanged`接口。`ReadOnlyObservableCollection<T>`的实例只能通过一个`ObservableCollection<T>`后台列表进行构建。尽管集合对调用者来说依然是只读的，但它们可以观察对后台列表其他地方的改变。

尽管通常情况下我建议使用接口作为API中方法的返回值，但特意公开`ReadOnlyCollection<T>`也是很有用的，它可以为调用者清楚地指明不能修改返回的集合。但仍需写明基础集合是否可以在其他地方修改，或是否为有效的常量。

B.3　字典

在框架中，字典的选择要比列表少得多。只有三个主流的非并发`IDictionary<TKey, TValue>`实现，此外还有`ExpandoObject`（第14章已介绍过）、`ConcurrentDictionary`（将在介绍其他并发集合时介绍）和`RouteValueDictionary`（用于路由Web请求，特别是在ASP.NET MVC中）也实现了该接口。

注意，字典的主要目的在于为值提供有效的键查找。

B.3.1　**`Dictionary<TKey, TValue>`**

如果没有特殊需求，`Dictionary<TKey, TValue>`将是字典的默认选择，就像`List<T>`是列表的默认实现一样。它使用了散列表，可以实现有效的查找（参见http://mng.bz/qTdH），虽然这意味着字典的效率取决于散列函数的优劣。可使用默认的散列和相等函数（调用键对象本身的`Equals`和`GetHashCode`），也可以在构造函数中指定`IEqualityComparer<TKey>`作为参数。

最简单的示例是用不区分大小写的字符串键实现字典，如代码清单B-1所示。

代码清单B-1　在字典中使用自定义键比较器

```
var comparer = StringComparer.OrdinalIgnoreCase;
var dict = new Dictionary<String, int>(comparer);
dict["TEST"] = 10;
Console.WriteLine(dict["test"]);                  <—— 输出10
```

尽管字典中的键必须唯一，但散列码并不需要如此。两个不等的键完全有可能拥有相同的散列码；这就是散列冲突（hash collision）[①]，尽管这多少会降低字典的效率，但却可以正常工作。如果键是易变的，并且散列码在插入后发生了改变，字典将会失败。易变的字典键总是一个坏主意，但如果确实不得不使用，则应确保在插入后不会改变。

① http://en.wikipedia.org/wiki/Collision_(computer_science)。——译者注

散列表的实现细节是没有规定的，可能会随时改变，但一个重要的方面可能会引起混淆：尽管Dictionary<TKey, TValue>有时可能会按顺序排列，但无法保证总是这样。如果向字典添加了若干项然后迭代，你会发现项的顺序与插入时相同，但请不要信以为真。有点不幸的是，刻意添加条目以维持排序的实现可能会很怪异，而碰巧自然扰乱了排序的实现则可能带来更少的混淆。

与List<T>一样，Dictionary<TKey, TValue>将条目保存在数组中，并在必要的时候进行扩充，且扩充的平摊复杂度为O(1)。如果散列合理，通过键访问的复杂度也为O(1)；而如果所有键的散列码都相等，由于要依次检查各个键是否相等，因此最终的复杂度为O(n)。在大多数实际场合中，这都不是问题。

B.3.2 **SortedList<TKey, TValue>**和**SortedDictionary<TKey, TValue>**

乍一看可能会以为名为SortedList<,>的类为列表，但实则不然。这两个类型都是字典，并且谁也没有实现IList<T>。如果取名为ListBackedSortedDictionary和TreeBacked-SortedDictionary可能更加贴切，但现在改已经来不及了。

这两个类有很多共同点：比较键时都使用IComparer<TKey>而不是IEquality-Comparer<TKey>，并且键是根据比较器排好序的。在查找值时，它们的性能均为O(log n)，并且都能执行二进制搜索。但它们的内部数据结构却迥然不同：SortedList<,>维护一个排序的条目数组，而SortedDictionary<,>则使用的是红黑树结构（参见维基百科条目http://mng.bz/K1S4）。这导致了插入和移除时间以及内存效率上的显著差异。如果要创建一个排序的字典，SortedList<,>将被有效地填充，想象一下保持List<T>排序的步骤，你会发现向列表末尾添加单项是廉价的（若忽略数组扩充的话将为O(1)），而随机添加项则是昂贵的，因为涉及复制已有项（最糟糕的情况是O(n)）。向SortedDictionary<,>中的平衡树添加项总是相当廉价（复杂度为O(log n)），但在堆上会为每个条目分配一个树节点，这将使开销和内存碎片比使用SortedList<,>键值条目的数组要更多。

这两种集合都使用单独的集合公开键和值，并且这两种情况下返回的集合都是活动的，因为它们将随着基础字典的改变而改变。但SortedList<,>公开的集合实现了IList<T>，因此可以使用排序的键索引有效地访问条目。

我不想因为谈论了这么多关于复杂度的内容而给你造成太大困扰。如果不是海量数据，则可不必担心所使用的实现。如果字典的条目数可能会很大，你应该仔细分析这两种集合的性能特点，然后决定使用哪一个。

B.3.3 **ReadOnlyDictionary<TKey, TValue>**

熟悉了B.2.5节中介绍的withReadOnlyCollection<T>后，ReadOnlyDictionary<TKey, TValue>应该也不会让你感到特别意外。ReadOnlyDictionary<TKey, TValue>也只是一个围绕已有集合（本例中指IDictionary<TKey, TValue>）的包装器而已，可隐藏显式接口实现后所有发生变化的操作，并且在调用时抛出NotSupportedException。

与只读列表相同，ReadOnlyDictionary<TKey, TValue>的确只是一个包装器；如果基础集合（传入构造函数的集合）发生变化，则这些修改内容可通过包装器显现出来。

B.4　集

在.NET 3.5之前，框架中根本没有公开集（set）集合。如果要在.NET 2.0中表示集，通常会使用Dictionary<,>，用集的项作为键，用假数据作为值。.NET3.5的HashSet <T>在一定程度上改变了这一局面，现在.NET 4还添加了SortedSet<T>和通用的ISet<T>接口。尽管在逻辑上，集接口应该只包含Add/Remove/Contains操作，但ISet<T>还指定了很多其他操作来控制集（ExceptWith、IntersectWith、SymmetricExceptWith和UnionWith）并在各种复杂条件下验证集（SetEquals、Overlaps、IsSubsetOf、IsSupersetOf、IsProperSubsetOf和IsProperSupersetOf）。所有这些方法的参数均为IEnumerable<T>而不是ISet<T>，这乍看上去会很奇怪，但却意味着集可以很自然地与LINQ进行交互。

B.4.1　**HashSet<T>**

HashSet<T>是不含值的Dictionary<,>。它们具有相同的性能特征，并且你也可以指定一个IEqualityComparer<T>来自定义项的比较。同样，HashSet<T>所维护的顺序也不一定就是值添加的顺序。

HashSet<T>添加了一个RemoveWhere方法，可以移除所有匹配给定谓词的条目。这可以在迭代时对集进行删减，而不必担心在迭代时不能修改集合的禁令。

B.4.2　**SortedSet<T>**（.NET 4）

就像HashSet<T>之于Dictionary<,>一样，SortedSet<T>是没有值的SortedDictionary<,>。它维护一个值的红黑树，添加、移除和包含检查（containment check）的复杂度为O(log n)。在对集进行迭代时，产生的是排序的值。

和HashSet<T>一样它也提供了RemoveWhere方法（尽管接口中没有），并且还提供了额外的属性（Min和Max）用来返回最小和最大值。一个比较有趣的方法是GetViewBetween，它返回介于原始集上下限之内（含上下限）的另一个SortedSet<T>。这是一个易变的活动视图——对于它的改变将反映到原始集上，反之亦然，如代码清单B-2所示。

代码清单B-2　通过视图观察排序集中的改变

```
var baseSet = new SortedSet<int> { 1, 5, 12, 20, 25 };
var view = baseSet.GetViewBetween(10, 20);
view.Add(14);
Console.WriteLine(baseSet.Count);                  <—— 输出6
foreach (int value in view)
{
   Console.WriteLine(value);                   <—— 输出12、14、20
}
```

尽管GetViewBetween很方便，却不是免费的午餐：为保持内部的一致性，对视图的操作可能比预期的更昂贵。尤其在访问视图的Count属性时，如果在上次遍历之后基础集发生了改变，操作的复杂度将为O(n)。所有强大的工具，都应该谨慎用之。

SortedSet<T>的最后一个特性是它公开了一个Reverse()方法，可以进行反序迭代。Enumerable.Reverse()没有使用该方法，而是缓冲了它调用的序列的内容。如果你知道要反序访问排序集，使用SortedSet<T>类型的表达式代替更通用的接口类型可能会更有用，因为可访问这个更高效的实现。

B.5 Queue<T>和Stack<T>

队列和栈是所有计算机科学课程的重要组成部分。它们有时分别指FIFO（先进先出）和LIFO（后进先出）结构。这两种数据结构的基本理念是相同的：向集合添加项，并在其他时候移除。所不同的是移除的顺序：队列就像排队进商店，排在第一位的将是第一个被接待的；栈就像一摞盘子，最后一个放在顶上的将是最先被取走的。队列和栈的一个常见用途是维护一个待处理的工作项清单。

正如LinkedList<T>一样，尽管可使用普通的集合接口方法来访问队列和栈，但我还是建议使用指定的类，这样代码会更加清晰。

B.5.1 Queue<T>

Queue<T>实现为一个环形缓冲区：本质上它维护一个数组，包含两个索引，分别用于记住下一个添加项和取出项的位置（slot）。如果添加索引追上了移除索引，所有内容将被复制到一个更大的数组中。

Queue<T>提供了Enqueue和Dequeue方法，用于添加和移除项。Peek方法用来查看下一个出队的项，而不会实际移除。Dequeue和Peek在操作空（empty）队列时都将抛出InvalidOperationException。对队列进行迭代时，产生的值的顺序与出队时一致。

B.5.2 Stack<T>

Stack<T>的实现比Queue<T>还简单——你可以把它想成是一个List<T>，只不过它还包含Push方法用于向列表末尾添加新项，Pop方法用于移除最后的项，以及Peek方法用于查看而不移除最后的项。同样，Pop和Peek在操作空（empty）栈时将抛出InvalidOperationException。对栈进行迭代时，产生的值的顺序与出栈时一致——即最近添加的值将率先返回。

B.6 并行集合（.NET 4）

作为.NET 4并行扩展的一部分，新的System.Collections.Concurrent命名空间中包含一些新的集合。它们被设计为在含有较少锁的多线程并发操作时是安全的。该命名空间下还包含三个用于对并发操作的集合进行分区的类，但在此我们不讨论它们。

B.6.1 IProducerConsumerCollection<T>和BlockingCollection<T>

IProducerConsumerCollection<T>被设计用于BlockingCollection<T>，有三个新的集合实现了该接口。在描述队列和栈时，我说过它们通常用于为稍后的处理存储工作项；生产者/消费者模式是一种并行执行这些工作项的方式。有时只有一个生产者线程创建工作，多个消费者线程执行工作项。在其他情况下，消费者也可以是生产者，例如，网络爬虫（crawler）处理一个Web页面时会发现更多的链接，供后续爬取。

IProducerConsumerCollection<T>是生产者/消费者模式中数据存储的抽象，BlockingCollection<T>以易用的方式包装该抽象，并提供了限制一次缓冲多少项的功能。BlockingCollection<T>假设没有东西会直接添加到包装的集合中，所有相关方都应该使用包装器来对工作项进行添加和移除。构造函数包含一个重载，不传入IProducerConsumer-Collection<T>参数，而使用ConcurrentQueue<T>作为后台存储。

IProducerConsumerCollection<T>只提供了三个特别有趣的方法：ToArray、TryAdd和TryTake。ToArray将当前集合内容复制到新的数组中，这个数组是集合在调用该方法时的快照。TryAdd和TryTake都遵循了标准的TryXXX模式，试图向集合添加或移除项，返回指明成功或失败的布尔值。它允许有效的失败模式，降低了对锁的需求。例如在Queue<T>中，要把“验证队列中是否有项”和“如果有项就进行出队操作”这两个操作合并为一个，就需要一个锁——否则Dequeue就可能抛出异常[①]。

BlockingCollection<T>包含一系列重载，允许指定超时和取消标记，可以在这些非阻塞方法之上提供阻塞行为。通常不需要直接使用BlockingCollection<T>或IProducerCon-sumerCollection<T>，你可以调用并行扩展中使用了这两个类的其他部分。但了解它们还是很有必要的，特别是在需要自定义行为的时候。

B.6.2 ConcurrentBag<T>、ConcurrentQueue<T>和ConcurrentStack<T>

框架自带了三个IProducerConsumerCollection<T>的实现。本质上，它们在获取项的顺序上有所不同；队列和栈与它们非并发等价类的行为一致，而ConcurrentBag<T>没有顺序保证。

它们都以线程安全的方式实现了IEnumerable<T>。GetEnumerator()返回的迭代器将对集合的快照进行迭代；迭代时可以修改集合，并且改变不会出现在迭代器中。这三个类都提供了与TryTake类似的TryPeek方法，不过不会从集合中移除值。与TryTake不同的是，IProducerConsumerCollection<T>中没有指定TryPeek方法。

B.6.3 ConcurrentDictionary<TKey, TValue>

ConcurrentDictionary<TKey, TValue>实现了标准的IDictionary<TKey, TValue>

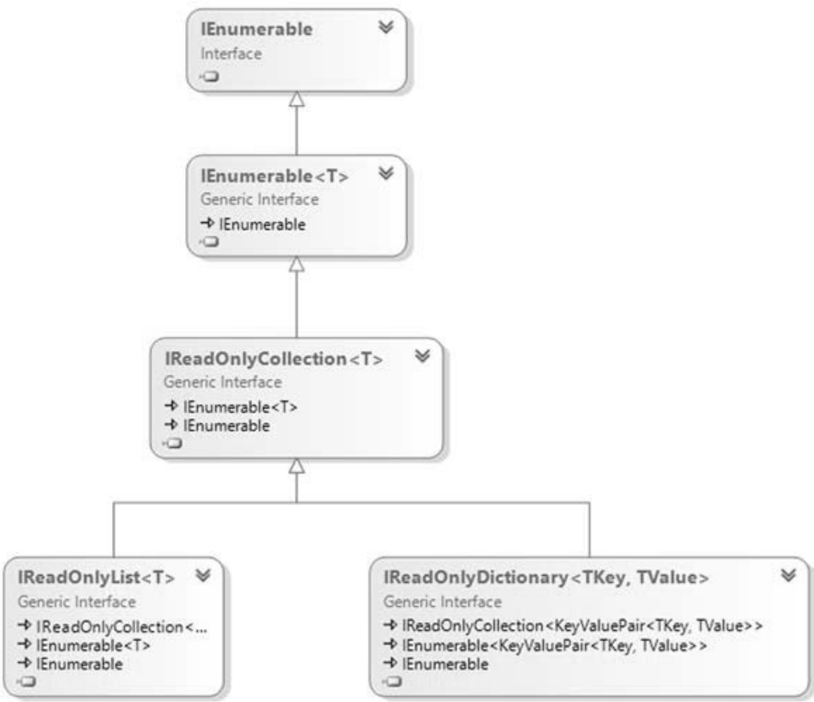
① 例如，当队列有且仅有一个项时，两个线程同时判断它是否有项，并且都返回true，这时其中一个线程先执行了出队操作，而另一个线程再执行出队操作时，由于队列已经空了，因此将抛出异常。——译者注

接口（但是所有的并发集合没有一个实现了`IList<T>`），本质上是一个线程安全的基于散列的字典。它支持并发的多线程读写和线程安全的迭代，不过与上节的三个集合不同，在迭代时对字典的修改，可能会也可能不会反映到迭代器上。

它不仅仅意味着线程安全的访问。普通的字典实现基本上可以通过索引器提供添加或更新，通过`Add`方法添加或抛出异常，但`ConcurrentDictionary<TKey, TValue>`提供了名副其实的大杂烩。你可以根据前一个值来更新与键关联的值；通过键获取值，如果该键事先不存在就添加；只有在值是你所期望的时候才有条件地更新；以及许多其他的可能性，所有这些行为都是原子的。在开始时都显得很难，但并行团队的Stephen Toub撰写了一篇博客，详细介绍了什么时候应该使用哪一个方法（参见http://mng.bz/WMdW）。

B.7 只读接口（.NET 4.5）

NET 4.5引入了三个新的集合接口，即`IReadOnlyCollection<T>`、`IReadOnlyList<T>`和`IReadOnlyDictionary<TKey, TValue>`。截至本书撰写之时，这些接口还没有得到广泛应用。尽管如此，还是有必要了解一下的，以便知道它们不是什么。图B-2展示了三个接口间以及和`IEnumerable`接口的关系。



图B-2 .NET 4.5的只读接口

如果觉得ReadOnlyCollection<T>的名字有点言过其实，那么这些接口则更加诡异。它们不仅允许其他代码对其进行修改，而且如果集合是可变的，甚至可以通过结合对象本身进行修改。例如，List<T>实现了IReadOnlyList<T>，但显然它并不是一个只读集合。

当然这并不是说这些接口没有用处。IReadOnlyCollection<T>和IReadOnlyList<T>对于T都是协变的，这与IEnumerable<T>类似，但还暴露了更多的操作。可惜IReadOnlyDictionary<TKey, TValue>对于两个类型参数都是不变的，因为它实现了IEnumerable<KeyValuePair<TKey, TValue>>，而KeyValuePair<TKey, TValue>是一个结构，本身就是不变的。此外，IReadOnlyList<T>的协变性意味着它不能暴露任何以T为参数的方法，如Contains和IndexOf。其最大的好处在于它暴露了一个索引器，通过索引来获取项。

目前我并没怎么使用过这些接口，但我相信它们在未来肯定会发挥重要作用。2012年底，微软在NuGet上发布了不可变集合的预览版，即Microsoft.Bcl.Immutable。BCL团队的博客文章（http://mng.bz/Xlqd）道出了更多细节，不过它基本上无须解释：不可变的集合和可冻结的集合（可变集合，在冻结后变为不可变集合）。当然，如果元素类型是可变的（如StringBuilder），那它也只能帮你到这了。但我依然为此兴奋不已，因为不可变性实在是太有用了。

B.8　小结

.NET Framework包含一系列丰富的集合（尽管对于集来说没那么丰富）[①]。它们随着框架的其他部分一起逐渐成长起来，尽管接下来的一段时间内，最常用的集合还应该是List<T>和Dictionary<TKey, TValue>。

当然未来还会有其他数据结构添加进来，但要在其好处与添加到核心框架中的代价之间做出权衡。也许未来我们会看到明确的基于树的API，而不是像现在这样使用树作为已有集合的实现细节。也许可以看到斐波纳契堆（Fibonacci heaps）、弱引用缓存等——但正如我们所看到的那样，对于开发者来说已经够多了，并且有信息过载的风险。

如果你的项目需要特殊的数据结构，可以上网找找开源实现；Wintellect的Power Collections作为内置集合的替代品，已经有很长的历史了（参见http://powercollections.codeplex.com）。但在大多数情况下，框架完全可以满足你的需求，希望本附录可以在创造性使用泛型集合方面扩展你的视野。

① 作者前面使用了a rich set of collecions，后面用了a rich collection of sets，分别表示丰富的集合和集。此处的中文无法体现原文这种对仗。——译者注

附录 C

版本总结

.NET的版本号有时可能会让人稀里糊涂。框架、运行时、Visual Studio和C#都包含各自的版本号。本附录是有关它们如何整合在一起以及各个版本主要特性的核心指南。对于每个概念，我都只会描述2.0及以上版本的特性；列出.NET 1.0和.NET 1.1的所有特性没有任何意义。

C.1　桌面框架的主版本

当开发者提到.NET的版本时，通常是指桌面框架的主版本。大多数情况下，框架发布时都会伴随Visual Studio的发布（或Visual Studio .NET，就如以2002和2003版本来命名）。例外情况是.NET 3.0，它实质上只是一组库（尽管这组库意义非凡）。Visual Studio 2005为这些新特性提供了一些扩展，不过Visual Studio 2008包含更多的支持。表C-1展示了框架的哪个部分的哪个版本在何时发布。

表C-1　桌面框架版本及其组件

日　　期	框　　架	Visual Studio	C#	CLR
2002年2月	1.0	2002	1.0	1.0
2003年4月	1.1	2003	1.2	1.1
2005年11月	2.0	2005	2.0	2.0
2006年11月	3.0	2005扩展	n/a	2.0
2007年11月	3.5	2008	3.0	2.0 SP1
2010年4月	4	2010	4.0	4.0（无3.0版本）
2012年8月	4.5	2012	5.0	4.0或4.5①

.NET 3.5发布时，.NET 2.0 SP1和.NET 3.0 SP1也同时发布，它们包含CLR和BCL的2.0 SP1。类似地，.NET 3.5 SP1发布的同时也发布了.NET 2.0 SP2和.NET 3.0 SP2。

Visual Studio 2008是第一个支持多目标（multitargeting）的版本，并可选择基于哪个框架

① 具体取决于个人观点，稍后会讲到相关内容。

版本进行构建。很多时候都可以在面向更早的版本时使用最新的C#特性——只要该特性是完全由编译器魔法实现的，并且不依赖于CLR或库即可。本书的网站上包含了更多这方面的信息（参见http://mng.bz/YpRB）。在一些情况下，如果某个特性不属于该框架，也有解决办法。值的注意的是，如果在Visual Studio 2008或2010中面向.NET 2.0（不可以是.NET 1.0或1.1）开发，则实际上面向的是相关的服务包（2.0 SP1或2.0 SP2）。这意味着如果你使用服务包中包含的新特性来构建代码（注意2.0 SP1中的`System.DateTimeOffset`），但运行的机器上安装的是.NET 2.0的原始版本，会导致失败。就我个人来说，我会在机器运行最新的服务包，最好是最新的完整框架版本。

C.2　C#语言特性

如果你通读了全书，应该可以自己编写本节的内容（我可以留下一堆空行让你填写，不过我还没这么懒）。一个不起眼的事实是：表C-1中的版本号1.2不是笔误。查看规范可知，微软确实跳过了C# 1.1，而在.NET 1.1的基础上发布了C# 1.2编译器。1.2版中的改动基本都很小，只有一个从长远来看意义重大的更改：只有C# 1.2及后续版本在翻译`foreach`循环时，会检查迭代器是否实现了`IDisposable`，并相应地进行释放。如我们所见，这个更改对于有资源需要清理的迭代器块来说至关重要。

无论如何，为了完整起见，下面列出了语言特性，以及可以查看详细内容的章节。

C.2.1　C# 2.0

C# 2的主要特性是泛型（参见第3章）、可空类型（第4章）、匿名方法及其他有关委托的增强（第5章）和迭代器块（第6章）。此外还包含一些小特性：分部类型、静态类、包含不同访问修饰符的属性的取值方法和赋值方法、命名空间别名、pragma指令以及固定大小的缓冲器，详细内容参见第7章。

C.2.2　C# 3.0

C# 3为LINQ而生，尽管很多特性在其他地方也很有用。自动属性、数组和局部变量的隐式类型、对象和集合的初始化程序以及匿名类型都在第8章中进行了介绍。Lambda表达式和表达式树（第9章）延伸了2.0中对委托所做的进展，扩展方法（第10章）构成了查询表达式（第11章）的最后一块拼图。分部方法仅出现在C# 3中，在第7章讲述分部类型时进行了介绍。

C.2.3　C# 4.0

C# 4中的特性旨在提高互操作性，但它不像C# 3.0那样一门心思为了LINQ。同样，对第13章的一些小特性（命名实参、可选参数、更好的COM交互、泛型可变性）和动态类型这个大特性（第14章）进行了相当清晰的划分。

C.2.4　C#5.0

第15章和第16章分别介绍了C# 5.0的异步特性和两个小特性（foreach变量捕获的变化和调用者信息特性）。尽管异步特性只引入了一个新的表达式，即async函数中的await，但却在很大程度上改变了执行模型。尽管C#团队做好了发布其他大型语言特性的准备（据我所知确实如此），我还是认为暂缓发布是个明智之举。重点在于C#社区要谨慎面对async/await，而这需要时间。

C.3　框架库的特性

我们无法以合理的方式列出框架的所有新特性。尤其是框架各个部分（Windows Forms、ASP.NET等）的各个版本除了核心基础类库之外，还包含自身的特性。我介绍了自认为最重要的特性。MSDN中有一个更全面的列表：http://mng.bz/6tiZ。

C.3.1　.NET 2.0

2.0库所支持的CLR和语言最重要的特性是泛型和可空类型。尽管可空类型不需要进行过多的修改，但某些从.NET 2.0以来一直存在的泛型集合及其反射API却需要相应地更新。

很多部分只进行了很小的修改，如支持压缩[①]、在SQL Server单个连接上的多活动结果集（Multiple Active Result Sets，MARS），以及很多静态的I/O辅助方法，如File.ReadAllText。公平地说，这些都不如对用户界面框架的改变重要。

ASP.NET新增了母版页、预编译功能以及很多新的控件。Windows Forms增加了TableLayoutPanel及类似的类，从而在布局能力上有了一个飞跃；通过双缓冲、新的数据绑定模型、ClickOnce部署等，进一步增强了性能。.NET 2.0引入的BackgroundWorker可以在多线程应用程序中轻松安全地更新UI，严格意义上它并不是Windows Forms的一部分，但在.NET 3.0的WPF到来之前，Windows Forms一直都是它主要的应用场景。

C.3.2　NET 3.0

.NET 3.0有点奇特，因为它是一个在CLR、语言和已有库方面都没有改变的“主”版本，而是由4个新的库组成。

- WPF是下一代用户界面框架；它是一场革命，而不仅仅是对Windows Forms的革新，尽管这两者可以共同存在。它跟Windows Forms是两种完全不同的模型，在本质上更倾向于组装式。Silverlight的用户界面基于WPF。
- WCF（Windows Communication Foundation）是构建面向服务的应用程序架构；它不会局限于单个协议，而是可以进行扩展，并且致力于统一现有的RPC类的通信管道，如远程处理。

① 指System.IO.Compression命名空间包含的对流进行基本压缩和解压缩的类。——译者注

- WF（Wondows Workflow Foundation）是用于构建工作流应用程序的系统。
- Windows CardSpace是一个安全识别系统。

这四个领域中，WPF和WCF已得到蓬勃发展，而WF和CardSpace似乎还未得到很好的推广。这并不是说后两种技术没有用，或以后不会变得很重要，只是在本书撰写之时还未普及而已。

C.3.3　.NET 3.5

.NET 3.5中最大的新特性是C# 3.0和VB 9所支持的LINQ。它包括LINQ to Objects、LINQ to SQL、LINQ to XML以及提供底层支持的表达式树。

其他方面也有一些重要的特性：在ASP.NET中可以更加简便地使用AJAX；WCF和WPF都在很大程度上得到了改进；引入了一个插件框架（`System.AddIn`）；新增了各种加密算法，等等。对于那些对并发和时间相关的API感兴趣的开发者，我有必要向你介绍`ReaderWriterLockSlim`和急需的`TimeZoneInfo`、`DateTimeOffset`类型。如果你使用.NET 3.5或更高的版本却仍旧到处依赖`DateTime`，你应该意识到除此之外还存在着更好的选择[1]。

.NET 3.5 SP1中最值得注意的库特性是Entity Framework及相关的ADO.NET技术，同时其他技术也得到了微小的改进。同样重要的是，.NET 3.5 SP1还引入了Client Profile——桌面.NET框架的缩减版，不包含很多用于服务器端开发的类库。这样就可以对只有客户端的应用程序进行小规模的部署。

C.3.4　.NET 4.0

长期以来，.NET 4库以各种不同的形式添加了不少内容。DLR是一个重要的部分，此外我们还在其他章节（简单）介绍了并行扩展。和前几版一样，用户界面也有了很大的改进，但为富客户端所做的改进主要集中在WPF，而不是Windows Forms。现有的核心API还进行了很多微调，以增加易用性，如`String.Join`现在接受`IEnumerable<T>`，而不再坚持只用字符串数组。这并不是什么重大改进，但如果它们能让每一位开发者轻松那么一点点，累积起来效果也是显著的。我们已经看到了现有的泛型接口和委托是如何具备协变性和逆变性的（如`IEnumerable<T>`变为`IEnumerable<outT>`，`Action<T>`变为`Action<in T>`），不过还有一些新的类型值得探索。

`System.Numeric`是为数值计算新增的命名空间。截至本书撰写之时，它只包含`BigInteger`和`Complex`类型，未来可能还会添加`BigDecimal`。`System`命名空间也新增了一些类型，如用于延迟初始化值的`Lazy<T>`，以及与第3章的`Pair<T1,T2>`类功能相同的`Tuple`泛型类家族，它最多达8个类型参数。`Tuple`还支持结构化比较，由`System.Collections`命名空间中的`IStructuralEquatable`和`IStructuralComparable`接口表示。尽管第12章中介绍的全部Reactive Extensions类都不属于.NET 4，但其核心接口`IObserver<T>`和`IObservable<T>`则位于

[1] 我个人感觉对复杂而有趣的时间和日期来说，这些支持并不够，因此我启动了Noda Time项目（参见https://code.google.com/p/noda-time/），但至少应使用`TimeZoneInfo`来简洁地表示一个不同于当地时区的时区。

`System`命名空间。我之所以把这些具体项提出来，是因为尽管像托管可扩展性框架（Managed Extensibility Framework，MEF）之类的新领域已经得到了广泛的关注，但还是容易忽视这些简单的类型。我希望你们把时间花在整个框架上，而不仅仅是那些外表光鲜的新玩意儿。

C.3.5　.NET 4.5

同样，驱动.NET 4.5变化的最大动力自然是异步。每一个你希望为异步的API都应该有异步版本：如果操作需要一定的耗时，就应该让它异步地执行。.NET 4的TPL进行了扩展（和优化），也有助于我们实现这一点。

.NET 4.5还包含一些其他的变化，要在此全部列出是不可能的。MSDN页面中列出的高亮部分（http://mng.bz/6tiZ）也比我想介绍的多。但大多变化都取决于所构建的项目，然而随着时间的推移，异步对于整个平台的颠覆，将会影响到每一个人。

C.4　运行时（CLR）特性

CLR的改变相对于库和语言的新特性来说，不太容易被广大开发者察觉。当然像泛型这样十分闪亮的特性会引起所有人的注意，但其他的就没这么明显了。至少在主版本方面，CLR的改变远没有语言或框架库频繁。

C.4.1　CLR 2.0

除了泛型，CLR还要作一处修改，以支持C# 2的新语言特性：即对第4章中介绍的可空值类型提供装箱和拆箱行为。

CLR 2.0还包括其他主要的更改。最重要的是支持64位处理器（x64和IA64），以及在SQL Server 2005中承载CLR的功能。SQL Server集成需要设计新的承载API，这样宿主机可以对CLR进行更多的控制，包括如何分配内存和线程。这使“刻苦”的主机可以确保运行于CLR中的代码不会危及关键进程中的其他部分，如数据库。

.NET 3.5包含CLR 2.0 SP1，.NET 3.5 SP1包含CLR 2.0 SP2。它们只包含相对较小的更改，如`DynamicMethod`中的代码可以访问其他类型的私有成员。CLR团队也一直寻找改善性能的途径，如改进垃圾回收、JIT以及启动时间等。

C.4.2　CLR 4.0

尽管CLR不需要任何改变就可以容纳DLR，但CLR团队仍然竭尽所能地做到更好，包括以下亮点。

- 改进了互操作封送性能和一致性的IL Stubs Everywhere（可查看http://mng.bz/56H6这篇有关.NET框架的博文，以了解更多细节）。
- 取代CLR 2.0中并发回收器的后台垃圾回收器。

- 替代CAS（Code Access Security），基于透明度概念且更加完善的安全模型。
- 用于支持C# 4内嵌PIA特性的类型等价。
- 在同一进程内同时执行不同的CLR。

.NET 4.5中的CLR包含一些改进，主要是围绕垃圾回收问题。可将其看作是一个次要（minor）版本。除了纯粹的性能提升，64位的CLR还支持`<gcAllowVeryLargeObjects>`配置选项，即使元素结构很大，也可以创建巨型数组。当然，前提是内存足够大。要解释这个版本号可能有点复杂。文档中描述CLR版本时使用的是CLR 4.5。但`Environment.Version`属性显示的是4.0。例如，现在运行版本是4.0.30319.18033，但有了服务包后，构建和版本号可能还会发生变化。

可查看.NET框架博客（http://blogs.msdn.com/b/dotnet），了解更多有关新特性的详细内容。

C.5　相关框架

计算机领域很少有模型能够以不变应万变，.NET也不例外。实际上，就连桌面框架现在也不再是单一版本：客户端配置（client profile）、32位和64位JIT、服务器和工作站，分别用于不同的任务。除此之外，还存在一些单独的框架，有自身的版本历史，可以针对不同的环境。

C.5.1　精简框架

精简框架（Compact Framework）最初的目的是用于运行Windows Mobile的移动设备。之后，它改变了目标，开始用于Xbox 360、Windows Phone7和Symbian S60。

精简框架的主要发布计划与桌面框架基本一致，不过并没有发布与.NET 3.0相对应的版本。有趣的是，最新的版本为3.7（用于某些Windows Mobile设备和WP7）。

精简框架的早期版本遗漏了一些十分重要的功能，不过社区的努力在很大程度上进行了填补。尽管它仍然是桌面框架的一个子集，但后来的版本弥补了很多重要缺陷。GUI层取决于具体的平台，例如，在Xbox 360上我们使用XNA，Windows Mobile支持Windows Forms，WP7支持XNA和Silverlight。运行于精简框架中的代码仍然可进行JIT编译和垃圾回收，尽管精简框架的回收器与桌面框架不同，前者不是分代式的。

C.5.2　Silverlight

Silverlight（http://silverlight.net/）主要目的是在浏览器或沙盒环境中（Silverlight 3）运行应用程序，它通常先通过浏览器进行安装。就此来说，它是Flash的天然竞争者；但就C#开发者来说，允许使用熟悉的库及语言来编写应用程序，具有明显的优势。Silverlight安装的是简化的CLR（称为CoreCLR，参见http://mng.bz/G32M）和类库，例如，不支持非泛型集合和Windows Forms。Silverlight的表示层基于WPF，但并不完全相同。它还支持深变焦（deep zoom）和自适应视频流等。

Silverlight 1于2007年9月发布，但只限于用XAML构造UI、用JavaScript编写逻辑。直到2008

年10月Silverlight 2发布，用C#编写Silverlight应用才成为现实。CoreCLR中的一些特性（在单进程中同时承载多个CLR和声明式透明度安全模型）现在已经成为桌面CLR 4.0中的一部分。它还包含了动态语言运行时的早期版本。

前进的步伐始终没有停止，2009年7月发布的Silverlight 3带来了更多的控件、更多的视频编解码器，以及离线和浏览器无关的应用程序。Silverlight团队继续着他们每9个月就发布一个新版本的习惯，于.NET 4发布的当周就发布了Silverlight 4，它包含了新功能长列表。Windows Phone 7支持Silverlight 3和一部分Silverlight 4的特性。之后发布的Windows Phone 7.1 SDK（以支持消费者手里标明7.5版本的手机），可支持大部分Silverlight 4特性。Windows Phone 7.x使用的均为精简框架CLR的演变版本。

Windows Phone 8可向后兼容支持Silverlight API，也可支持新的Windows Phone Runtime API，该API更接近于Windows Store应用程序所使用的WinRT API。此外，Windows Phone 8使用的不是精简框架，而是CoreCLR。

对于日后开发而言，Silverlight已不会再有所发展。尽管很多开发者仍在使用，但已经不会再推出新版本了。不过Silverlight开发者对WinRT肯定很熟悉。微软试图实现从Silverlight应用到Windows Store应用的平滑过渡。

C.5.3　微框架

微框架（Micro Framework，参见http://mng.bz/D9qy）是.NET的一个十分微小的实现，可运行于受限的设备上。它不支持泛型，是解释型的而非JIT编译，并且发布了有限的类，但却包含了围绕WPF构建的展示层。为了节省空间，你只需要部署实际需要的那部分框架——最小情况下可以仅占用390 KB。这显然是一个很小的生态环境，但能够在嵌入式设备中编写托管代码有巨大的吸引力。它不适用于所有情况，比如它不是一个实时系统，但在其适用的地方，可以大幅提高开发者的生产力。

微框架的发布历史没有跟随桌面框架的步伐：它最先于2004年出现在SPOT手表中，1.0版本则于2006年发布。从那之后，它快速地更新换代。2009年11月19日发布了微框架4.0版。让人欢欣鼓舞的是，该版本的主要部分基于Apache 2.0许可进行了开源。由于种种原因，有些库，像TCP/IP栈和密码的实现，仍然是封闭的。对于这些库，可以下载其二进制形式的具体架构。

C.5.4　Windows Runtime (WinRT)

WinRT并不是.NET的另一个版本，而是Windows 8引入的全新Windows平台。它要在x86和ARM处理器架构上提供一个沙盒环境，并支持多语言——主要是.NET中的C#和VB、C++/CX（专门用于WinRT的新式C++）和JavaScript。它是一个非托管的API，但却与.NET紧密集成。因此C#和VB.NET开发者可以和C++/CX以及JavaScript开发者一样，使用相同的API，而无须像Win32使用Windows Forms那样，构建一个包装的API。这些API从设计之初就融入了异步思想。使用异步来面向WinRT开发应用程序，已经成了一种最常规的方式。

Windows 8还是一个年轻的操作系统，我们还没能看到它的成功。开发者希望创建的Windows 8应用也能在传统平台上运行，但显然微软认为WinRT是客户端开发的未来方向。以后，Windows Phone API和Windows Store API可能会越来越接近。

C.6　小结

面对如此繁多的组件，如此芜杂的版本，我们很容易迷糊，也更容易让别人迷糊。作为最后的忠告（尽管说是最后，但很难将这些忠告按深度和意义排序），我建议你在与别人交流这个话题时，要尽可能明确。如果你使用的不是桌面框架，就直说。如果要引用一个版本号，要指明确切的内容，“3.0”可能意味着使用C# 2.0和.NET 3.0，或C# 3.0和.NET 3.5。不管别的书怎么写，在看完本书后，你绝对没有任何理由说你正在使用的是C# 3.5或C# 4.5，除非你是故意气我。



深入理解C#（第3版）

“学习C#语言特性的最佳资源。”

——Andy Kirsch，Venga

“本书使我的C#水平更上一层楼。”

——Dustin Laine，Code Harvest

“每一位.NET开发人员都至少应该阅读一遍的案头必备图书。”

——Dror Helper，Better Place

“本书无疑是我读过的最佳C#参考书。”

——Jon Parish，Datasift

本书是世界顶级技术专家“十年磨一剑”的经典之作，在C#和.NET领域享有盛誉。与其他泛泛介绍C#的书籍不同，本书深度探究C#的特性，并结合技术发展，引领读者深入C#的时空。作者从语言设计的动机出发，介绍支持这些特性的核心概念。作者将新的语言特性放在C#语言发展的背景之上，用极富实际意义的示例，向读者展示编写代码和设计解决方案的最佳方式。同时作者将多年的C#开发经验与读者分享，读者可咀其精华、免走弯路，使程序设计水平更上一层楼。

本书在第2版的基础上全面调整了C#语言的细节，改写了随着技术的发展已经不再适用的内容，并全面介绍了C# 5新增的大特性——异步，以及两个小特性，延续了读者期望的高标准。

MANNING

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议　计算机/程序设计/C#

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-34642-1

定价：99.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks